Teaching programming with objects

Geoffrey G. Roy and Jocelyn Armarego
School of Engineering Science
Murdoch University

This paper describes a new tool (*P-Coder*) that can assist in the teaching of Object– Oriented (O-O) programming concepts to novices. It builds on a pseudocode model that focuses on the most basic computational principles of sequence, iteration, selection and recursion. These are developed within an O-O framework that provides several views of the model: a Design View where the user interacts with the model to design the program (algorithms and structure); a Class View where the class structure can be visualised; a Code View where the generated code can be inspected, compiled and executed; and an Object View where objects can be instantiated from the defined classes.

P-Coder provides a complete environment to demonstrate and implement many O-O concepts for novice users. There is a focus on developing an understanding of basic principles through a graphic/text based pseudocode notation representing the key program elements in a tree structured diagram.

Users are able to design and build complete programs, compile and execute them in the normal way. Interactive object instantiation is also supported, allowing users to create, inspect and manipulate objects in their runtime state. This provides a useful perspective in teaching where the distinctions between Classes and Objects are often left confused.

Keywords: pseudocode, programming, novices, object-oriented

Introduction

Teaching programming has been recognised as a difficult task, even from the early days of digital computation. The structured approach (Dijkstra, 1972) offered the first really formal approach once it was recognised that it was necessary to improve the quality of software from design/development, performance and maintenance perspectives. In more recent times Object-Oriented (O-O) approaches have opened up a new set of issues (Kölling, 1999; Northrop, 1992; Osborne, 1992), in particular just how and when O-O concepts should be taught. The purists will argue that O-O should be taught from day one (Adams & Frens, 2003; Cooper, Dann, & Pausch, 2003; Duke, Salzman, Burmeister, Poon, & Murray, 2000; Kölling, Koch, & Rosenberg, 1995), while others (Decker & Hirshfield, 1994) take a contrary view. Lewis (2000) presents a number of myths surrounding the use and application of O-O technologies and its associated pedagogies. One of these says that: “Object-orientation and procedure concepts are mutually exclusive”.

Lewis argues that the key underlying principles in O-O, including modularity and encapsulation, are equally well applied to procedural programming, though the implementing languages do not often enforce them. Lewis’ fifth myth is concerned with control structures: “Control structures and objects are in contention”. He argues that the low level computational primitives are neither procedural nor object-orientated, but they are orthogonal to the higher level design (O-O or procedural) concepts. As a result these lower level concepts must be taught in their own right and cannot be deferred, or neglected, in the early phases of a teaching programme. It has been claimed that O-O concepts more closely align to natural human problem solving processes (Decker & Hirshfield, 1993; DeClue, 1996) and thus are potentially more readily appreciated by the novice programmer. This argument often appears as a justification for teaching O-O principles early rather than later in an educational programme. Duke et al (2000) offer a more penetrating view. They argue that while some elements of O-O architectures (e.g. message passing and class re-use) should be taught up front, much of the detail of object construction can be deferred until later in the course. Duke et al (2000 p 84) also state that:

> … a first subject needs to emphasise the logical relationships between lower level concepts and how to capture these within the code.

The first author's own teaching experience supports this view, and in particular Duke et al (2000 p 84)'s conclusion that:

> ... the most significant difficulty facing beginners: using basic programming constructs such as while-loops and boolean expressions to capture a system's internal logic.

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 principles (i.e. sequence, iteration, selection and recursion) are understood. Unfortunately, regardless of which paradigm is adopted these basic principles are often not well understood.

The approach to be presented here fits within an O-O paradigm, with Java as a target language vehicle. It offers, however, an early focus on the basic computational processes. The *P-Coder* tool, which has been developed for this purpose, encourages (requires) that students develop their first appreciation of computation from these principles. While concepts of classes (their attributes and operations) are described early, most actual programming is done within a highly constrained environment. This hides much of the complexity from the novice so they can focus on developing knowledge of the principles. Over time the "training wheels" are gradually removed.

Most approaches to teaching programming skills require an early introduction of language syntax. This is necessary to allow even the simplest of program to successfully compile and execute. *P-Coder* provides considerable help in this phase of learning by restricting the available language elements. It also provides considerable support to assist the novice programmer to identify and construct their very first program statements. The early focus is on computational principles. As the novice's experience grows additional capabilities can be enabled thus providing a more open programming environment.

The *P-Coder* framework

The *P-Coder* model is the central element of the *P-Coder* framework shown in Figure 1. It supports a number of Views of the model and a variety of transformations that makes the software being developed accessible (visible, testable and usable). An introduction to *P-Coder* and its pseudocode based modelling paradigm is given elsewhere (Armarego & Roy, 2004).

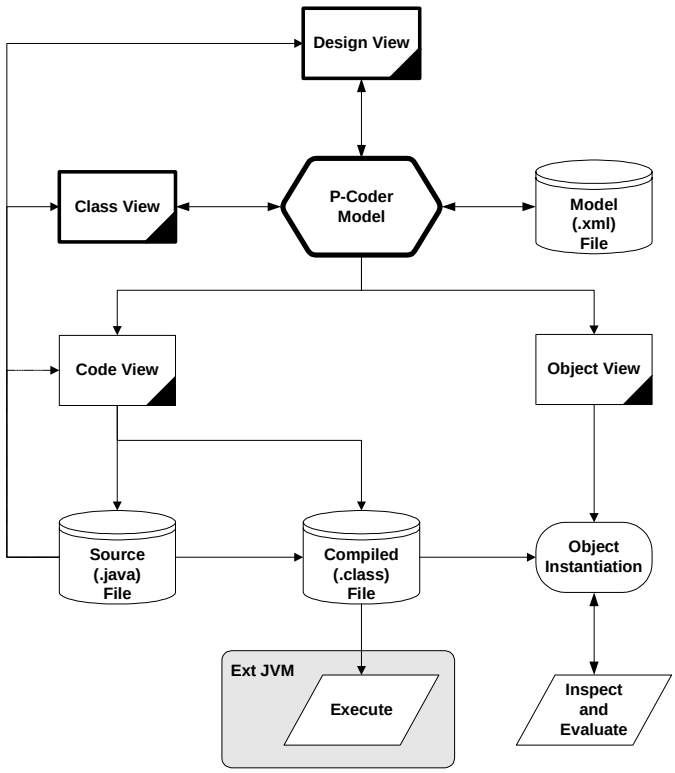

Figure 1: The *P-Coder* modelling framework

The environment supports four Views of the program being developed:

1. **The Design View**: where the model is created, visualised and edited, with a focus on algorithm and structure
2. **The Class View**: where the class structure can be displayed and edited showing the class associations and dependencies
3. **The Code View**: where the actual program code is created, made readable, compiled and executed as required
4. **The Object View**: where the compiled classes can be instantiated as objects that can be inspected and evaluated in an interactive fashion.

Each of these provides an alternative way of looking at the same model, and thus offers the potential of explaining the underlying model from different perspectives.

Normally the Design View provides the primary specification interface where the user views, builds and edits the model. The user can also create and edit program elements in the Class View. It is also possible (with some limitations) to construct a model (described as a skeleton model) from a set of existing source files. The skeleton model will contain the primary structure of the program (down to the method definition level). The *P-Coder* model is based on an abstract tree structure using a pseudocode notation combining both text and graphics as shown in Figure 2. Here we have a program (partially exposed) to manipulate and display vectors showing the pseudocode notation in the Design View.

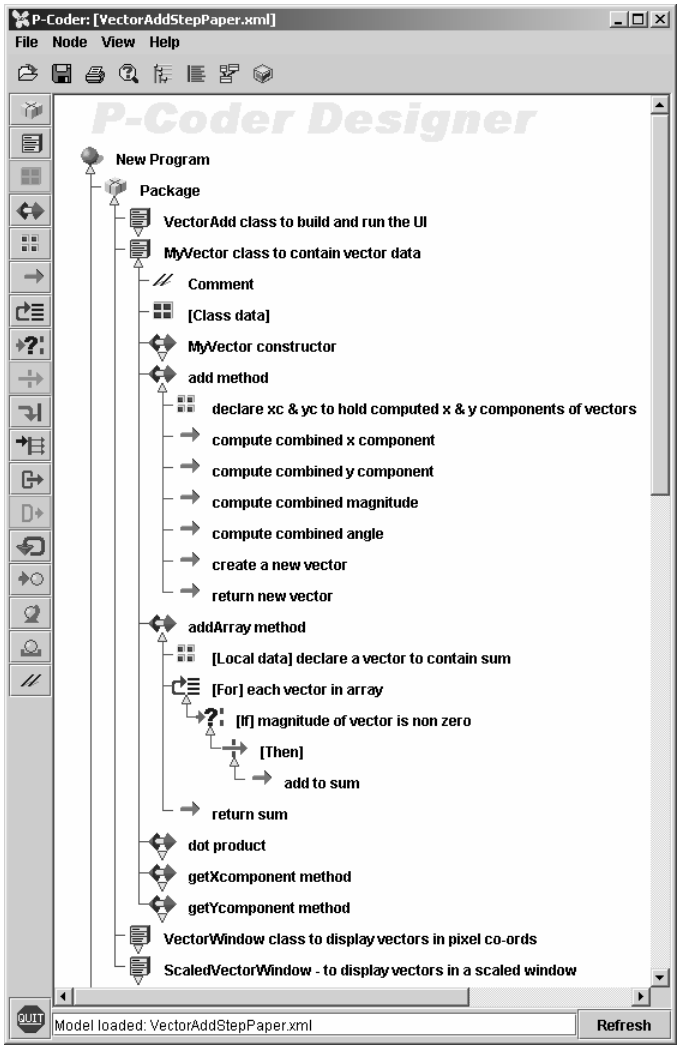

Figure 2: The Design View in *P-Coder*

The *P-Coder* tools offers a graphical/text based pseudocode notation using a tree structure to describe the computational process. At the top level, nodes in the model denote Packages and Classes. Classes contain class attributes, and operations. Operations contain the computational primitives of sequence, iteration, selection and recursion. The icons placed on each node are indicative of their semantics, while the added note qualifies or clarifies the meaning for the program being developed. The notes are free text and form part of the documentation in the code (but not the actual code).

Within a Method the computation proceeds down the tree, taking each right branch in turn until all available branches of the tree have been explored. At the higher levels the order of nodes (Packages, Classes and Methods) is not significant, but the same tree structure is maintained for consistency. Nodes can be rolled up or out as required to control the complexity of the display and to allow attention to be focussed on relevant parts of the model. A full range of interactive editing capabilities is also supported. The complete *P-Coder* model is externalised as an XML formatted file.

From this model it is possible to build much of the actual code (and certainly its structure). Some additional code segments must be added to complete the program. These are provided in node Details Dialogs that are attached to each node in the model, like that shown in Figure 3.

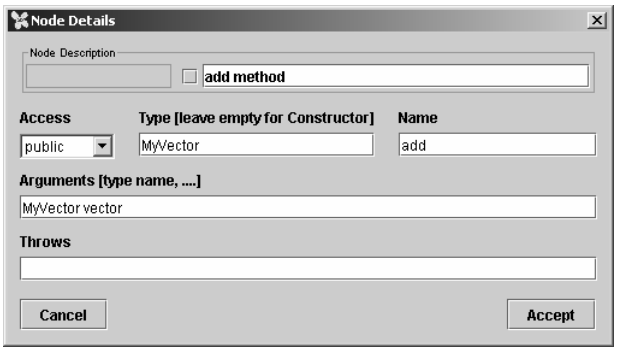

Figure 3: An example node details in *P-Coder*

Figure 4(a) shows a small section of generated code from the Code View. It includes the pseudocode notes (as comments with a special notation), the computational structure of the program, and the content from the relevant Details Dialogs. Keeping the pseudocode notes integrated with the code ensures a degree of literate programming (Knuth, 1984) is achieved. If required, the display of the notes can be suppressed, as shown in Figure 4(b), where only the raw code is shown. Explicit comments can be added to the model also, as comment nodes in the model. Typically, novice users are not permitted to edit the code directly - all editing is done in the Designer View. This ensures that the Code View is always consistent with the actual model definition.

Even before completing all the code segments it is possible to see much of the structure of the program in the Class View. Once the classes, their attributes and operation signatures are defined, all the information is in place to display the Class View as shown in Figure 5. Here we can see the “extends” relations and the “uses” relations between the four classes in the package. Each class box in this diagram can be expanded to show the attributes and operations within the class, and their respective properties as shown in Figure 6 (just the *MyVector* Class that defines the basic vector definitions is shown here). The class field definitions are shown together with the constructor and the class method signatures using a UML-styled notation.

The association types that can be displayed in the Class View are:

- “extends” relations: to describe inheritance relationships between classes
- “uses” relations: to describe where one class instantiates objects from another class
- “implements” relations: to describe interface implementations.

The Class View thus provides a clear visual description of the model in an O-O sense, by showing the structure of each class (attributes and operations) and the associations between classes. Changes in either the designer view or Class View are immediately reflected in the other view.

```
// -------------------------------------------------------------
// addArray method
// -------------------------------------------------------------
public MyVector addArray(MyVector [] array)
{
    /**< [Local data] declare a vector to contain sum >**/
    /**< sum will contain sum of vectors in array >**/
    MyVector sum = this;
    /**< [For] each vector in array >**/
    for(int i=0; i < array.length; i++)
    {
        /**< [If] magnitude of vector is non zero >**/
        if(array[i].mag != 0.0)
        /**< [Then] >**/
        {
            /**< add to sum >**/
            sum = sum.add(array[i]);
        }
    }
    /**< return sum >**/
    return sum;
}

/**< dot product >**/
// -------------------------------------------------------------
// dot product
// -------------------------------------------------------------
public double dotProduct(MyVector v)
{
    /**< [Local data]  >**/
    /**< angle between two vectors >**/
    double angle;
    /**< the dot product >**/
    double product;
    /**< compute angle >**/
    angle = v.angle - this.angle;
    /**< compute dot product >**/
    product = this.mag*v.mag*Math.cos(Math.toRadians(angle));
    /**< return product >**/
    return product;
}
```

```
// -------------------------------------------------------------
// addArray method
// -------------------------------------------------------------
public MyVector addArray(MyVector [] array)
{
    MyVector sum = this;
    for(int i=0; i < array.length; i++)
    {
        if(array[i].mag != 0.0)
        {
            sum = sum.add(array[i]);
        }
    }
    return sum;
}

// -------------------------------------------------------------
// dot product
// -------------------------------------------------------------
public double dotProduct(MyVector v)
{
    double angle;
    double product;
    angle = v.angle - this.angle;
    product = this.mag*v.mag*Math.cos(Math.toRadians(angle));
    return product;
}
```

(a) (b)

Figure 4: The code (partial) for the VectorAdd problem:(a) with pseudocode notes, (b) without pseudocode notes

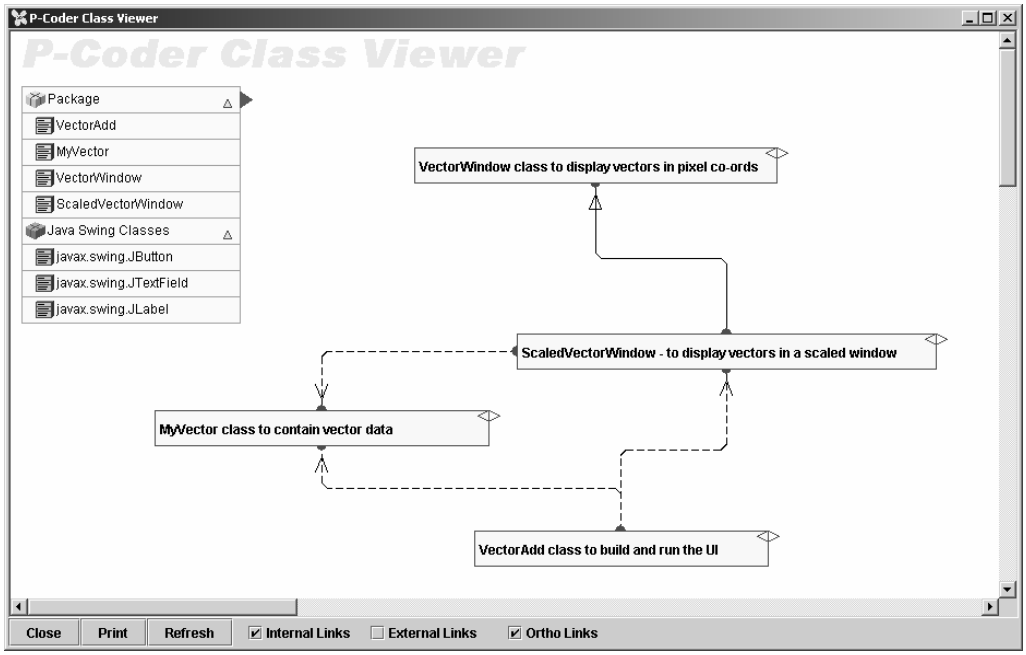

Figure 5: The Class View

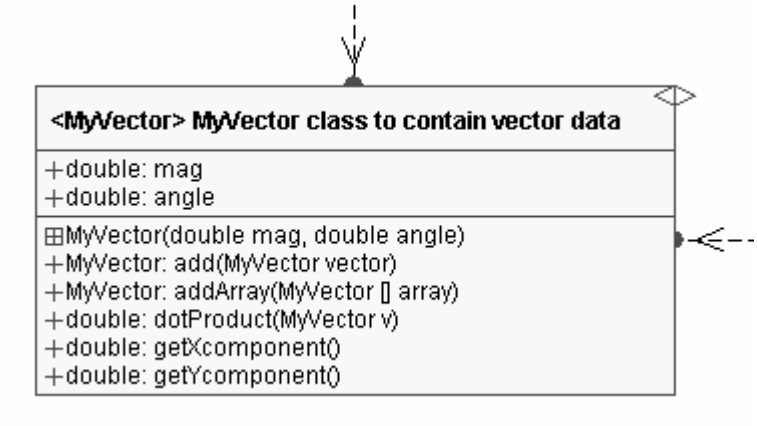

Figure 6: The expanded view of the MyVector class

The Class View is important as a means of displaying and explaining the associations that exist between classes within the model. It also provides a precursor to later studies in software requirements and architectures where professional CASE (computer Aided Software Engineering) tools and environments are used.

Dealing with objects

When working with novices, the basic computational processes are introduced in the Design View. While the Class View offers a view of the relationships between classes, it is the concept of the object that is often most difficult to explain. To assist in this aspect of teaching it is possible to provide object instantiation capabilities that can highlight these relationships.

Rosenberg and Kölling (1997), and more recently in relation to the *BlueJ* development environment Kölling and Rosenberg (2002), also argue that class descriptions by themselves are not adequate to test (and explain) the behaviour of the objects instantiated at runtime. They propose that instance creation capabilities are necessary to allow effective testing to be undertaken. For novices, this approach also has the great advantage of providing a view into the behaviour of the program, and its objects, that it not otherwise possible. This approach has been adopted here.

In *P-Coder* the Object View provides these facilities:

- To instantiate objects from classes: this allows the user to create one or more Objects from a Class that is included within the model
- To inspect objects, i.e. examine their fields and methods: this allows the user to observe the properties of these objects. The names and values of Class fields can be inspected and the names and signatures of Methods of the object are also visible
- To manipulate field values: items of Class data can have their values changed
- To execute Methods, passing arguments and capturing return values (and Objects) as required: i.e. the user can request a Method to evaluate using the Class data, and any required argument values, and then see the results of the computation and the returned values. For primitive data types the returned values are typically shown to the user. For compound types, and if a matching Class is included in the model, the user can capture the Object and save it in the Object View. This new Object can then be inspected.

In this way the user can see a concrete view of the Object and how it relates to the Class from which it is realised. The following examples demonstrate these features.

The Object View (see Figure 7) shows each Class in the model, and some other external classes. External classes are typically subsets of the Java (and other) class libraries that can be included if they are used within the model. In this figure, the *MyVector* Class has been selected; its fields and methods are then shown in the panels on the right. Selecting the constructor and then providing values for the arguments, and giving the new Object a name, will create an object based on this Class. Once created, the Object appears in the Object View to the right of its associated Class (as shown in Figure 8). Further objects can be created using the same Class constructor.

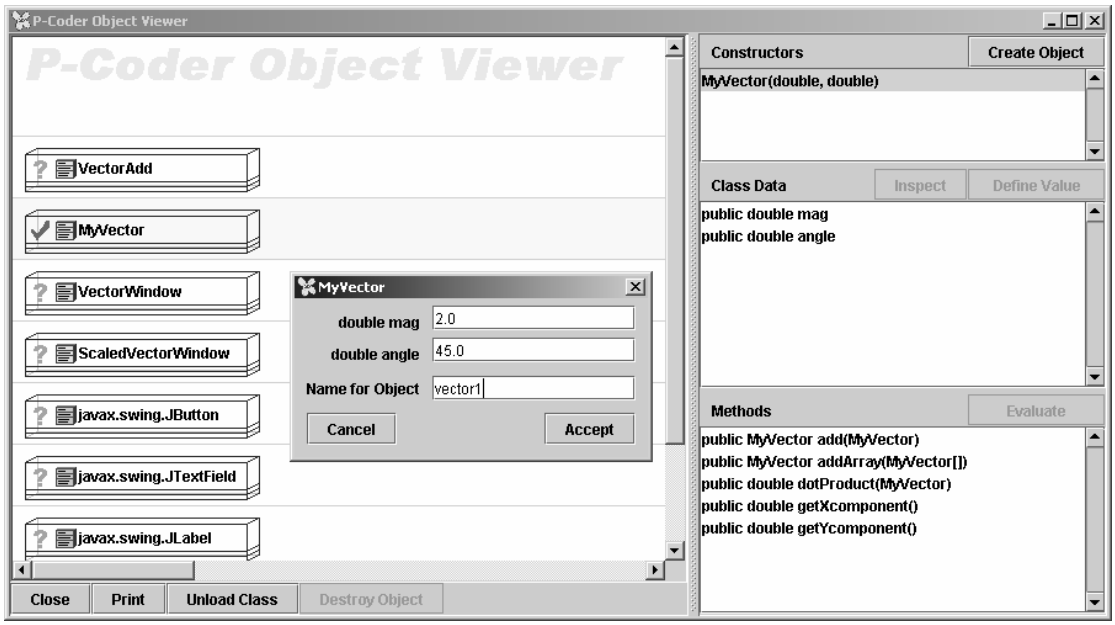

Figure 7: Instantiating an object

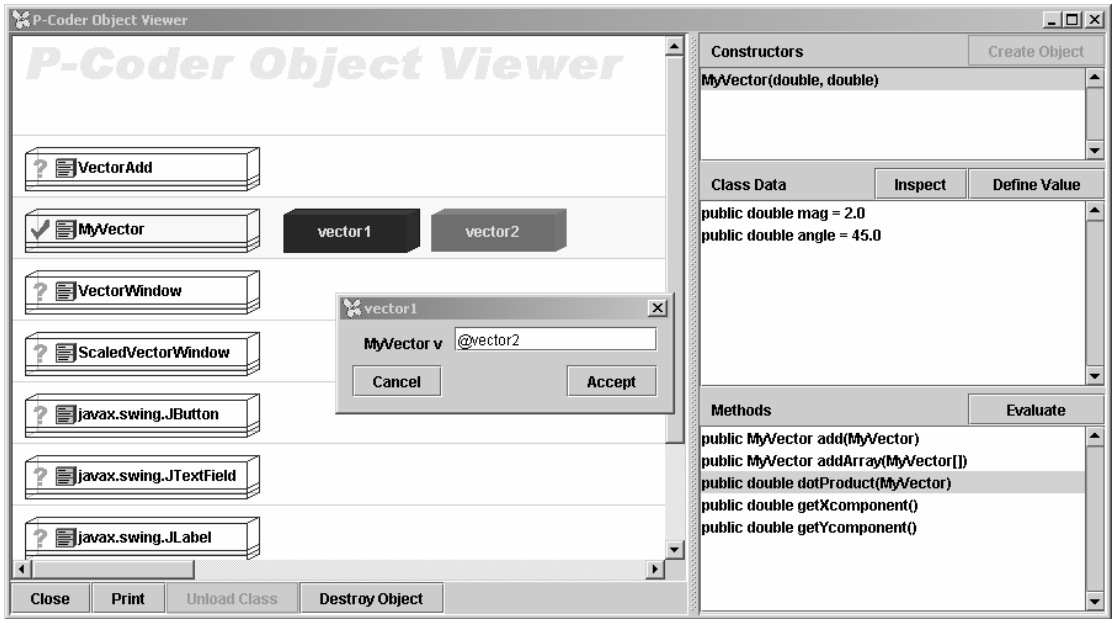

Figure 8: Evaluating the *dotProduct()* method of *vector1* and *vector2*

In Figure 8 we have selected the *vector1* Object then evaluated its *dotProduct()* Method with *vector2* used as an argument for the method. The returned result is then shown to the user (as shown in Figure 9). If we choose the *add()* Method of *vector1*, with *vector2* as an argument, the returned result is a *MyVector* type Object. Since the model knows about this type of Class, the returned Object can be captured (named *vector3*) and saved as shown in Figure 10. Selecting this new Object will show its field values (*mag* and *angle*) in the Class Data panel.

Where an Object is created from a Class that inherits properties from a parent Class, the class hierarchy can be explored as shown in Figure 11. In this case the *ScaledVectorWindow* Class has been selected. This Class is a child of the *VectorWindow* Class which is in turn sub-classed from *JFrame*, and so on. In Figure 11 the complete class hierarchy is displayed. For the selected Class, the parent classes provide fields and methods through inheritance. These can be loaded incrementally from the "Load" buttons, with the inherited fields and methods being added to the panels on the right. In this way it is possible to

explore the class hierarchy, identifying, inspecting data and evaluating methods as desired as the parent classes are incrementally loaded.

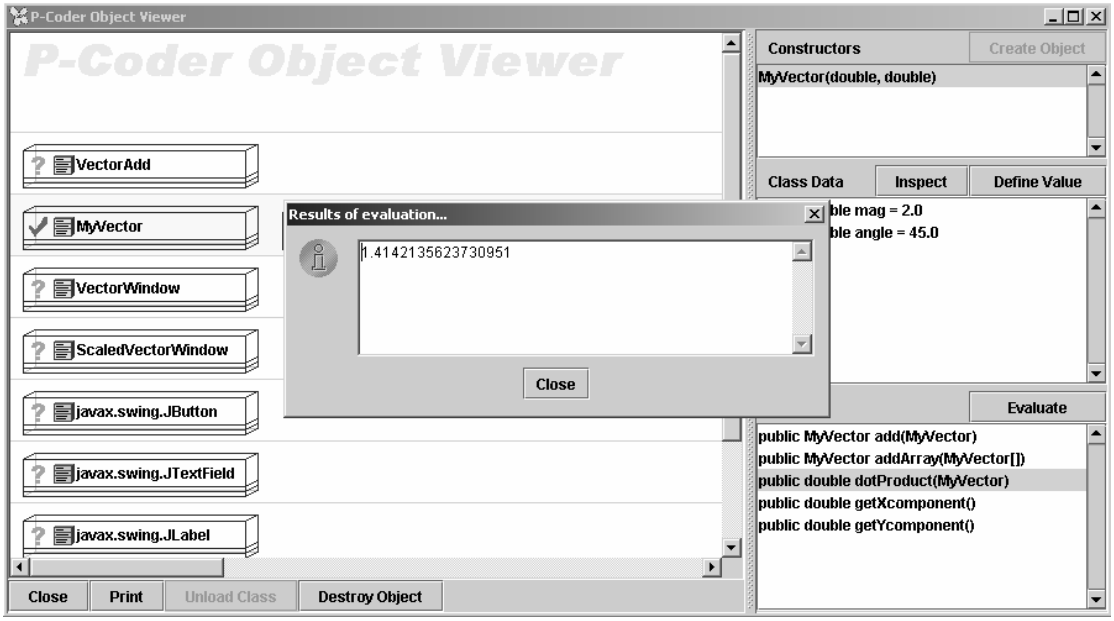

Figure 9: The results of evaluating the *dotProduct()* method

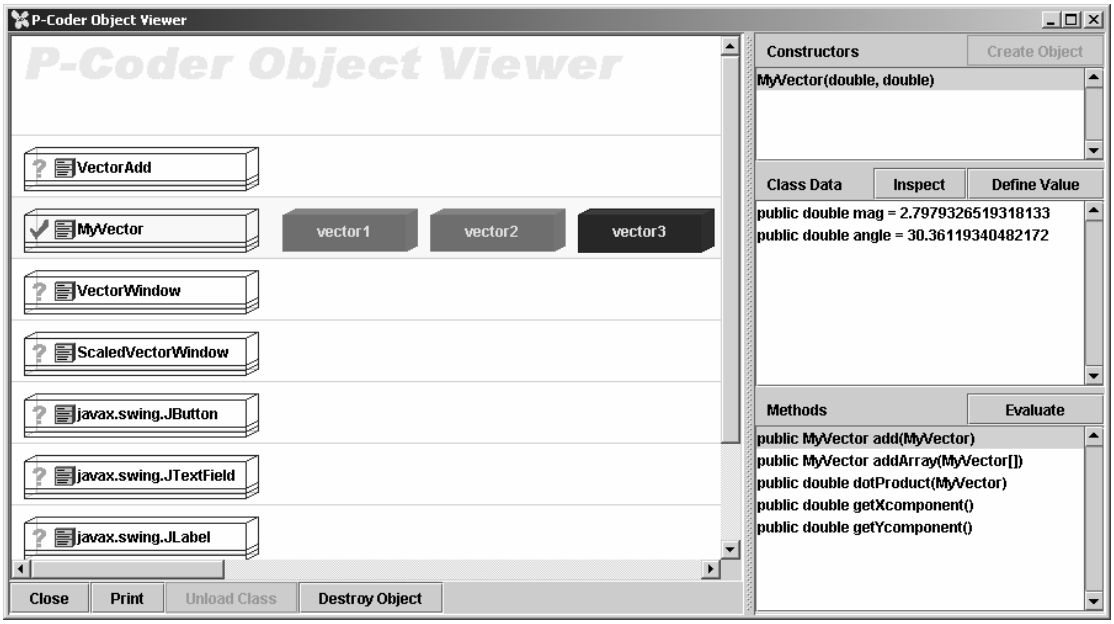

Figure 10: Inspecting the fields

For novices it may be necessary to limit the class hierarchy that is made visible, as the full Java class hierarchies are typically too complex (for the novice) to understand. *P-Coder* provides parameter settings to facilitate this.

Configuring *P-Coder* for teaching

P-Coder has a range of configuration options specifically designed to facilitate teaching. They include:

- Progressive access to the various model Views. For example restricting access to the Designer View for early beginners, then adding the Code View or Object View, to provide extended capabilities as learning proceeds
- Editing the Code View is normally disabled
- Access to the full Java class hierarchy in the Object View can be restricted
- A range of stylistic aspects can be controlled through parameters; including selected colours, some user interface behaviours, and strict adherence to UML conventions
- Activity logging on a user by user basis for progress monitoring and analysis of learning styles and processes
- Template models can be created to contain pre-set program elements with replaceable arguments (like Program and Class names, key variables etc) that can be user specified when the template first loaded.

If configured in a teaching laboratory, the properties file can be located on a file server (read only) so that the instructor can vary the settings during the progress of a course of study, thus providing additional functionality in a timely fashion.

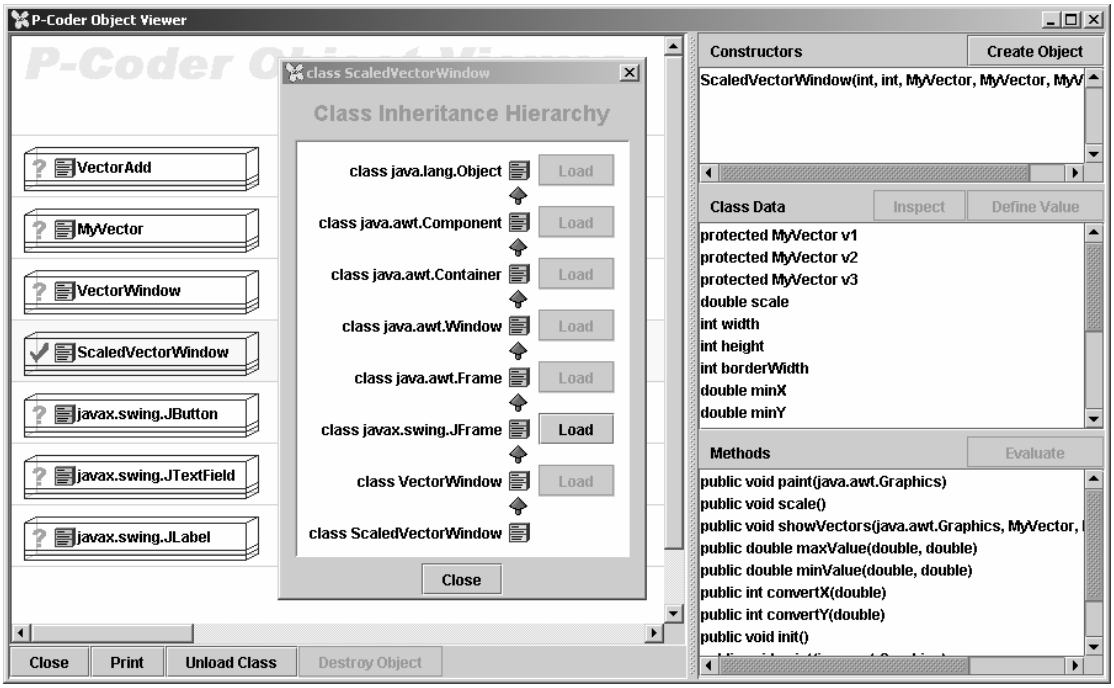

Figure 11: Exploring the class hierarchy

Summary

The *P-Coder* tool described here offers a range of capabilities in support of teaching O-O programming and design principles. The underlying model can provide a complete description of a program, or part of a program, including both low level (the basic computational elements) and higher level structures that are reflective of an O-O approach. Together these can provide an effective approach to teaching. This is achieved by providing a progressive exposure to syntax elements while focussing on computational principles, and also by facilitating access to the internals of objects at run-tine. This brings a degree of concreteness to what is otherwise rather abstract, especially the distinctions between classes and objects that often confuse novices.

P-Coder is currently implemented to just produce Java code. It does not support the full Java language, some of the more esoteric syntax forms are not implemented but this is unlikely to affect novices. *P-Coder* is not intended for the production of large scale software systems: it has been tested only on modest scale exercises (a few hundred lines of code). This is thought to be perfectly adequate for most early teaching exercises. Currently we use *BlueJ* (Kölling & Rosenberg, 1996) for this purpose. *BlueJ*

offers some of the capabilities of *P-Coder* (especially for interactive Object instantiation), but builds on a more traditional text based approach to programming.

While *P-Coder* is intended for teaching novices it would be interesting to explore its use in the development of more complex algorithms and program architectures for more experienced users. It is freely available from the authors for evaluation and educational use.

References

Adams, J. & Frens, J. (2003). *Object centered design for Java: Teaching OOD in CS-1.* Paper presented at the SIGCSE'03, Reno, Nevada, 19-23 February.

Armarego, J. & Roy, G. G. (2004). Teaching design principles in software design. *Beyond the Comfort Zone: Proceedings ASCILITE Conference*, Perth, 5-8 December. http://www.ascilite.org.au/conferences/perth04/procs/armarego.html

Cooper, S., Dann, W. & Pausch, R. (2003). Teaching objects-first in introductory computer science. Paper presented at the SIGCSE'03, Reno, Nevada, 19-23 February.

Decker, R. & Hirshfield, S. (1993). Top-down teaching: Object-oriented programming in CS1. Paper presented at the SIGCSE, Indianapolis.

Decker, R. & Hirshfield, S. (1994). The top 10 reasons why object-oriented programming can't be taught in CS 1. Paper presented at the SIGCSE'94, Phoenix, March.

DeClue, T. (1996). Object-orientation and the principles of learning theory: a new look at problems and benefits. Paper presented at the SIGCES'96, Philadelphia.

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. *Proceedings Australian Conference on Computer Science Education*, Melbourne.

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2), 97-111.

Kölling, M. (1999). The problem of teaching object-oriented programming, Part II: Environments. *Journal of Object-Oriented Programming*, 11(9), 6-12. [verified 26 Oct 2004] http://www.mip.sdu.dk/~mik/papers/oo-environments.pdf

Kölling, M., Koch, B. & Rosenberg, J. (1995). Requirements for a first year object-oriented teaching language. *Proceedings of 26th SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tennessee, March. [verified 26 Oct 2004] http://www.mip.sdu.dk/~mik/papers/requirements.pdf

Kölling, M. & Rosenberg, J. (1996). Blue - a language for teaching object-oriented programming. *Proceedings 27th SIGCSE Symposium on Computer Science Education*, Philadelphia. [verified 26 Oct 2004] http://www.mip.sdu.dk/~mik/papers/blue-paper.pdf

Kölling, M. & Rosenberg, J. (2002). *BlueJ - The Hitch-Hikers Guide to Object Orientation* (No. 2002 No 2): The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark. [verified 26 Oct 2004] http://www.mip.sdu.dk/~mik/papers/hitch-hiker.pdf

Lewis, J. (2000). Myths about object-orientation and its pedagogy. Paper presented SIGCSE 2000, Austin Texas.

Northrop, L. M. (1992). Finding an educational perspective for object-oriented development. Paper presented at the OOPSLA'92, Vancouver, 5-10 December.

Osborne, M. (1992). The role of object-oriented technology in the undergraduate computer science curriculum. Paper presented at OOPSLA'92, Vancouver, 5-10 December.

Rosenberg, J. & Kölling, M. (1997). Testing object-oriented programs: Making it simple. Paper presented at SIGCSE'97, California.

Please cite as: Roy, G.G. & Armarego, J. (2004). Teaching programming with objects. In R. Atkinson, C. McBeath, D. Jonas-Dwyer & R. Phillips (Eds), *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference* (pp. 811-820). Perth, 5-8 December. http://www.ascilite.org.au/conferences/perth04/procs/roy.html