Sense before syntax: a path to a deeper understanding of objects

Rob Griffiths, Simon Holland, Marion Edwards
Computing Department, The Open University, Walton Hall,
Milton Keynes, England MK7 6AA
r.w.griffiths@open.ac.uk, s.holland@open.ac.uk, medwards@nildram.co.uk

Abstract: This paper describes how we have successful adapted a principled pedagogy of objects first and progressive disclosure, originally developed for teaching objects concepts through the vehicle of a pure object language, to the teaching of object concepts using Java. We employ a cognitive science viewpoint to distinguish between, and sequence accordingly, two different aspects of learning Java. We focus initially on fundamental aspects of the object model of computation, which are simple, consistent, meaningful, and hence relatively stable in memory. Aspects of the Java syntax and semantics which are contingent or arbitrary, and hence unstable in long-term memory, are deferred until after students have acquired a secure conceptual model. We use three principal techniques to assist students in acquiring programming experience of fundamental concepts relatively un-distracted by contingent detail. These measures are: interactive microworlds that allow accurate visualisation of central object concepts; a Java scripting environment that minimises the amount of syntax required, but which allows students to interact with and inspect 'live' objects in the microworlds; and an explicitly object-oriented (if verbose) programming style that reinforces object-oriented concepts. Dealing with Java-specific design peculiarities is thus deferred until students have a stable conceptual model on which to scaffold a deeper understanding of objects.

Keywords: Java, microworld, objects first, progressive disclosure, OUWorkspace, BlueJ, scripting environment, object-oriented, Smalltalk, cognitive science.

1. Introduction

After thirty or more years experience, it may sometimes appear that there is very little genuinely new to be said about teaching object concepts to undergraduates. We argue that, to the contrary, there is plenty of room for new teaching insights to arise, for example by the application of new findings from areas such as cognitive science.

Equally, it is sometimes assumed that, for purposes of teaching object concepts, differences between object-oriented languages are minimal. Again, we will argue that this is not the case. Pure object languages such as Smalltalk that use a single consistent conceptual metaphor for computation, can be understood using much simpler cognitive structures than hybrid languages such as Java, which mix several conceptual metaphors inconsistently (Mortensen, 2001). Such simplicity makes simpler teaching strategies possible and makes it relatively straightforward to focus on fundamental concepts. The purpose of this paper is to investigate the degree to which some, if not all, of the pedagogical benefits afforded by pure object languages can be retained when teaching object concepts using hybrid languages, given an appropriately designed teaching strategy.

2. Background

The institutional backdrop to this work is the replacement of The Open University's highly successful course, M206: *Computing an Object-oriented Approach* (Woodman et al, 1998; Holland et al, 1997). This 60 point Smalltalk-based course won a prestigious BCS IT Award, was recognised for its innovation by attaining Design Council Millennium Product status, and attracted some 35,000 students in its presentation lifetime. With the introduction by the Open University of named degrees and with this previous course coming to the end of its presentation lifetime (2005) it was decided to replace it with two 30 point courses, one to teach Object-oriented Analysis and design (designated M256) and one to teach fundamental object-oriented programming principles (designated M255).

This latter course, M255, is the subject of this paper. Initially it was planned that M255 should continue the strategy of teaching object-oriented concepts using Smalltalk, but early in its design, a Departmental level decision based principally on marketing factors was taken to switch the main computing language for undergraduate teaching from

Smalltalk to Java, in order to better meet issues such as name recognition by students, and to more directly address student perception of employability issues. This posed the development team with a substantial problem; how to retain as many as possible benefits of a carefully designed and proven teaching strategy based on simplicity, consistency, and a clear conceptual model of computation, when switching to a hybrid language such as Java, which implements objects in a partial and irregular way (Bates, 2004).

As the purpose of the course is not to teach the minutiae of any particular language but rather to teach fundamental object-oriented programming concepts and skills transferable to any object-oriented language, we looked for ways to focus on fundamental aspects of the object model of computation, which are simple, consistent and meaningful, while deferring an emphasis on syntactic detail until students had a stable conceptual model against which the detail could be related. We found three principal measures to facilitate this in Java. The three measures were:

- Open-ended interactive microworlds that allow accurate visualisation of object references, message sending, state change and specialisation.
- A scripting environment for Java that minimises the amount of syntax that students initially need, but which allows them to create, interact with and inspect the state of 'live' objects that are automatically displayed in a graphical window.
- An explicitly object-oriented (if verbose) programming style that reinforces object-oriented concepts.

We will now deal with these measures in turn.

3. Microworlds

To provide a way of visualising, interacting with, and reasoning about concrete examples of object concepts, we designed a series of graphical microworlds concerning frogs and other amphibians. These microworlds allow the visible actions and state of amphibians to be controlled in two parallel ways – on the one hand via buttons and menus, and in parallel by sending messages to the amphibians using Java statements via a code pane. This duality reflects in a concrete form the heart of the object model of computation, which may be viewed as being based on a metaphor between objects and computers, and a recursion on this metaphor, viewing computation as built from networks of simpler computations collaborating together (Kay, 1993).

In particular, the amphibian microworlds model the behaviour of instances of the classes `Frog` and `Toad` and of a subclass of `Frog`, `HoverFrog`. As the name 'hoverfrog' implies, the classes are deliberately designed to be cartoon-like rather than realistic, and to be both visually and conceptually memorable. So for example, in the cartoon-like amphibian microworld, hoverfrogs may be positioned by students at arbitrary heights on the y axis, whereas simpler amphibians such as frogs, may be asked to hop only from stone to stone along the x-axis. This playful approach to abstracting state and behaviour is intended to help demystify the processes of abstraction and modelling. The simplicity and memorability is intended to give students a reference set of easy-to-memorise and eventually fully analysed examples to use as a portable personal resource throughout the course, able to illustrate the full range of object concepts.

Students interact with these microworlds at the very beginning of the course before they have seen any Java code. As already outlined, the microworlds are concrete cartoon-like worlds consisting of frogs and various other amphibians (two variations of the microworlds are shown in figures 1 and 2). For the purposes of the microworlds, frogs can be made to move their position and change their colour.

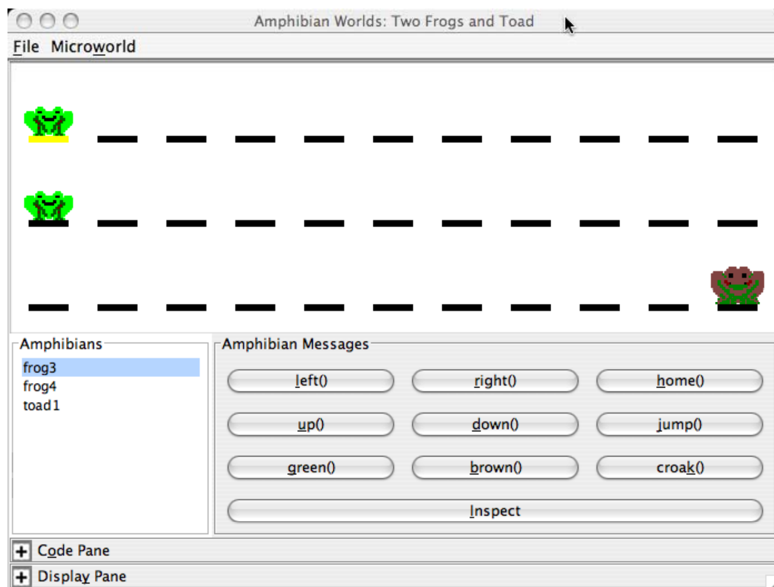

Figure 1

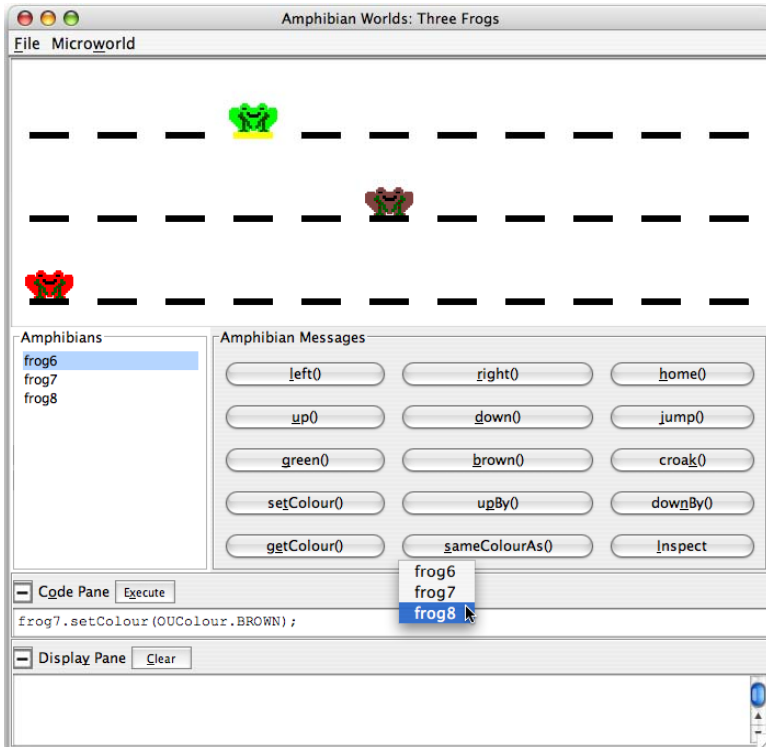

Figure 2

Via buttons, students can look at the state of frogs, send messages to them, see how they behave in response, see how this affects their state and look at how a message to one frog may in some cases cause a frog to send a message to another frog (sameColourAs() button in figure 2). As the students progress through the microworlds, more of the protocol of the amphibian objects, and the mechanisms used in their interactions are progressively exposed.

These microwords are also the vehicle by which students learn the syntax for writing message-sends (method invocations). Each microworld has a Code Pane in which they can write and execute Java statements (as shown in figure 2). By opening up a microworld's code pane they can write statements that can do everything that pressing buttons can do such as `frog1.right();` and `frog3.sameColourAs(hoverFrog2);`.

As already touched on, these microworlds have been devised to reveal fundamental object concepts including object reference, state change, polymorphism, specialisation and abstraction. The microworlds that students encounter are already populated with existing amphibians, but later in the course, students create new amphibians of various kinds which can be displayed in a graphical window.

Moving on to the interactive visualisation of object state and behaviour, figure 1 above shows a microworld which contains objects of the classes `Frog` and `Toad`. These two classes have identical attributes – `position` and `colour` – and identical message protocols, such as `green()`, `brown()`, `home()`, `right()` and `left()`, which respectively set the receiving object's `colour` to green, or brown, change its position to the "home" position and move left or right. Students select references to any of the objects in the microworld from a regular scrolling list and use buttons to send the corresponding messages. This simple user interface not only allows straightforward message sending to be visualized, it allows more abstract notions such as polymorphism to demonstrated; for example, when a frog is selected and the home() button is clicked (resulting in the message `home()` being sent) the receiving frog moves to the leftmost position, but if a toad has been selected, and so receives the message `home()`, it moves to the *rightmost* position – the "home" position for toads. This microworld also allows us to introduce the notion of class; frogs and toads do not behave identically to the same protocol, leading students to notions of different classes and different interfaces.

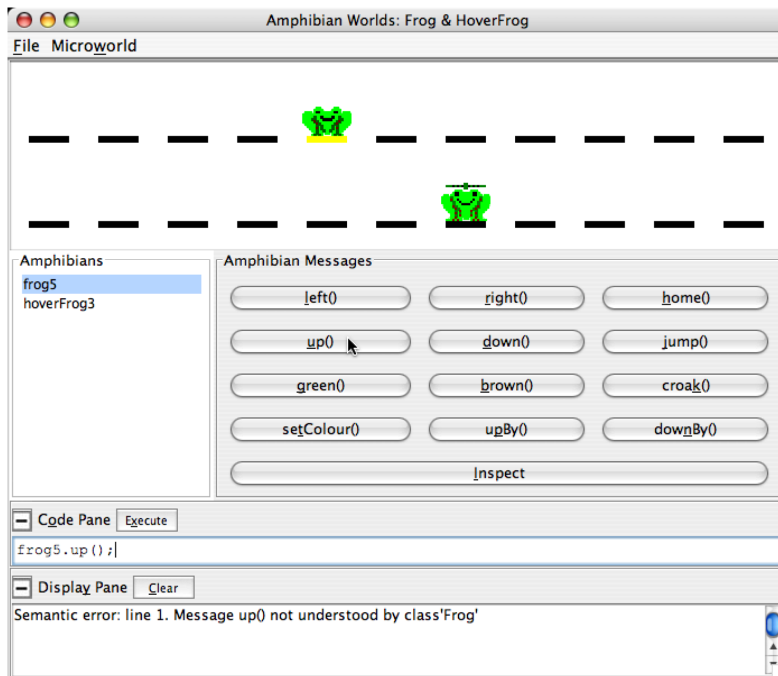

Figure 3

Figure 3 shows a microworld with a `Frog` object and a `Hoverfrog` object where students discover that instances of `Hoverfrog` understand all the messages sent by all the buttons

in the microworld. The same is not true of frogs and toads. When the messages `up()` or `down()` are sent to the frog object, the Display Pane opens up and a message informing the user that an error has occurred is displayed. Further inspection of `Frog` and `Hoverfrog` objects (figures 4 & 5) reveals that `Hoverfrog` objects have an additional instance variable – `height`.

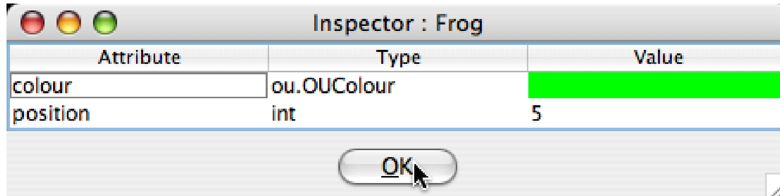

Figure 4

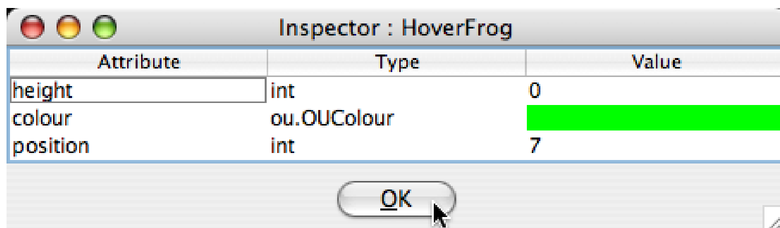

Figure 5

Through this exploration students are guided to discover that a hoverfrog has everything a frog object has but an extra attribute and an extended protocol – conceptually setting the scene to explore the fact that the `HoverFrog` class is a subclass of the `Frog` class. Once inheritance has been explicitly taught, students redesign these classes (`Frog`, `HoverFrog` and `Toad`) to be concrete subclasses of an abstract class `Amphibian`.

In figure 5 the inspector shows the state of a `HoverFrog` object. The inspector for an object always has three columns that list: the object's attributes, the types of those attributes and the values of those attributes. The inspectors are diving inspectors, therefore double-clicking on the colour row will reveal the state of the `OUColour` object as shown in Figure 6.

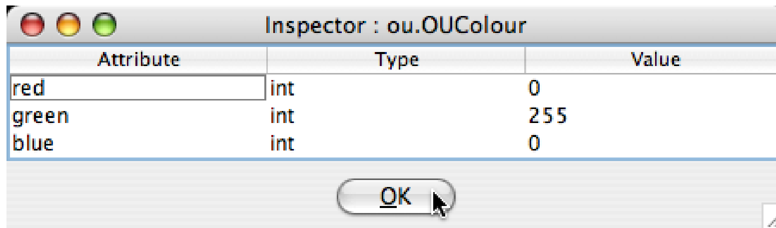

Figure 6

This, although not made explicit until later, reveals that fundamentally all objects (in Java at least) are composed of primitive types.

In the very initial stages, through exploration of these microworlds students quickly learn the following key ideas before getting to grips with the Java language:

- *Messages* – the only way to get an object to do anything is to send it a message
- *References* – to send a message to some object you need a way to refer to it.
- *Attributes* – by observing the results of sending messages to amphibian objects students discover that frogs and toads have the attributes of colour and position and hoverfrogs have the additional attribute height.
- *Class* – objects of the same class have the same attributes and the same behaviour
- *Inheritance* – objects that can do everything that another object can do – and them some more, are likely to be a instances of some subclass.

After learning the basic ideas about objects through exploring the microworlds, students move on to using a Java Integrated Development Environment (IDE). The IDE chosen was BlueJ. We chose this IDE as it has an extremely simple user interface, was specifically developed for teaching Java and is platform independent. Excellent though BlueJ is, we required a more flexible and expressive parallelism between interactively interpreted Java and graphical windows than was available in BlueJ. For this reason, we developed an extension to the environment called the OUWorkspace – where, very shortly, the same key ideas bulleted above are then explored in detail using sequences of messages executed in the OUWorkspace. This we describe in the next section.

4. The OUWorkspace

In a traditional Java course the very first thing that a student does is to write (or more probably copy) a completely static class as shown below:

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

Straight away students are faced with understanding (or perhaps not) the structure of a class file, the delimiters '`{`' and '`}`', the purpose and structure of the `main()` method, how to declare an array of strings and the reserved words: `public`, `static` and `void` – which at such a point in their study is information overload. They then have to compile the class and then finally execute the program (probably from the command line). More importantly the code has very little to do with objects. The only object created by the program is the literal string `"Hello World!"` and the only message in the code is `println()` sent to `out`. The BlueJ IDE (Kölling et al, 2003) does much better than this, however from our experience of developing an integrated Smalltalk learning environment (Woodman et al, 1999) we wished to develop a simpler solution better suited for distance learning where students have limited contact with tutors, and better suited to the teaching strategy outlined above. This involved developing the OUWorkspace.

The OUWorkspace is a scripting environment for Java built as an extension to BlueJ. It is opened from within BlueJ by selecting Tools | OUWorkspace. When opened it is configured to work with the currently open BlueJ project allowing the creation and manipulation of instances of the classes defined in that project. In addition the OUWorkspace has access to many of the standard Java classes and the classes in the course supplied OU Class Library. If no BlueJ project is open the OUWorkspace only has access to the standard Java classes and the classes in the OU Class Library. The fact that all these classes are in scope to the OUWorkspace means that we can defer another bit of syntax: the import statement.

The OUWorkspace (see figure 7) contains three panes labelled 'Code Pane', 'Display Pane' and 'Variables'. The Code Pane is used to declare variables, enter and execute Java statements. To execute the statements the user must first highlight them and then

select the Action | Execute Selected menu option or the Execute Selected option on the Code Pane's popup menu. The Display Pane is where any textual output relating to those executed statements, including error messages, is displayed. The list pane labelled Variables holds a list of the currently declared variables in this case `hoppyHeight` and `hoppy`.

If an error is detected when the selected code is executed an error message will be shown in the Display Pane. An error message is identified as a syntax error, a semantic error or an exception. If more than one line of code has been executed the error message includes the line number of the code containing the error. *This line number is relative to the highlighted code rather than all the code currently in the Code Pane.*

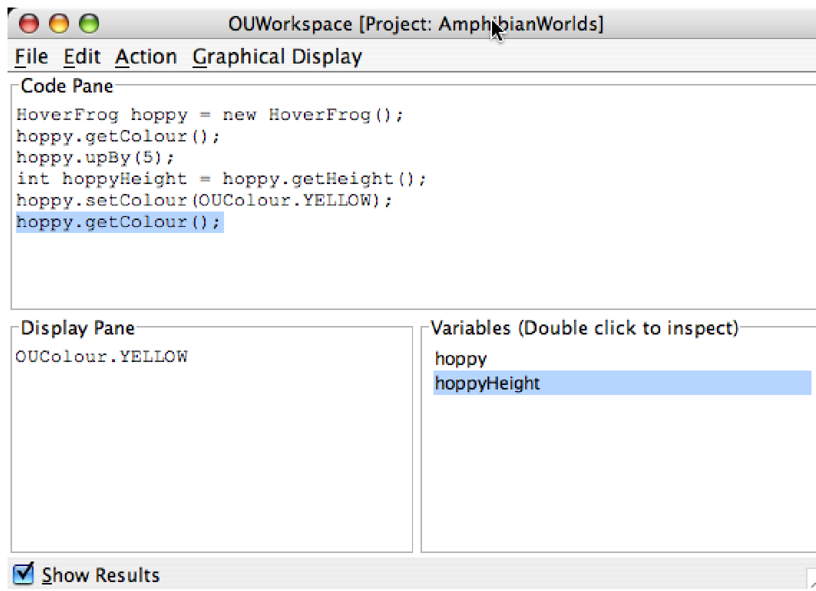

Figure 7

With the Show Results check box checked and if the last expression in a statement returns a value (either an object or a primitive) the textual representation of that value will be displayed in the Display Pane (as shown in figure 7). If the Show Results check box is not checked only the results of `System.out.println()` statements are shown in the Display Pane (figure 8).

Code Pane

```
for (int verseNumber = 2; verseNumber <= 10; verseNumber++)
{
  System.out.println(verseNumber
              + " men went to mow, went to mow a meadow,");
  System.out.println(verseNumber
              + " men went to mow, went to mow a meadow,");
  for (int men = verseNumber; men > 1; men--)
  {
    System.out.print(men + " men, ");
  }
  System.out.println("one man and his dog,");
  System.out.println("Went to mow a meadow.");
}
```

Display Pane

```
7 men, 6 men, 5 men, 4 men, 3 men, 2 men, one man and his dog,
Went to mow a meadow.
8 men went to mow, went to mow a meadow,
8 men went to mow, went to mow a meadow,
8 men, 7 men, 6 men, 5 men, 4 men, 3 men, 2 men, one man and his dog,
Went to mow a meadow.
9 men went to mow, went to mow a meadow,
9 men went to mow, went to mow a meadow,
9 men, 8 men, 7 men, 6 men, 5 men, 4 men, 3 men, 2 men, one man and his dog,
Went to mow a meadow.
10 men went to mow, went to mow a meadow,
10 men went to mow, went to mow a meadow,
10 men, 9 men, 8 men, 7 men, 6 men, 5 men, 4 men, 3 men, 2 men, one man and his dog,
Went to mow a meadow.
```

Show Results

Figure 8

If the currently opened BlueJ project includes classes whose instances can be displayed graphically (at present we support amphibians and shape classes), then a Graphical Display menu appears in the OUWorkspace's menu bar from which a graphical window can be opened. Figure 9 shows BlueJ with an open project that contains all the classes in the Amphibian hierarchy, the OUWorkspace and a graphical window capable of displaying amphibians.

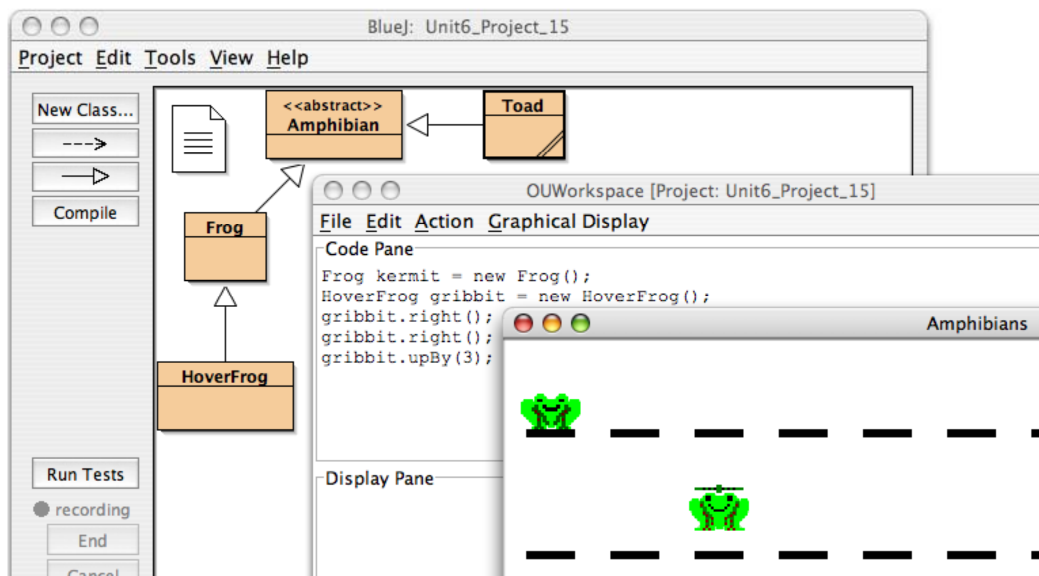

Figure 9

Any `Amphibian` object created in the OUWorkspace and assigned to a variable will immediately appear in the graphical window, as the domain of that window is the pool of variables declared in the OUWorkspace and any variables that reference objects of the correct type will have their graphical representation displayed. Any message-sends to amphibian objects in the OUWorkspace will therefore be visually demonstrated. Students subclass the existing classes in the `Amphibian` hierarchy and any objects of these subclasses that they create and assign to a variable in the OUWorkspace will automatically be visible in the graphical window too, exhibiting whatever behaviour students choose to give them.

Objects created in the OUWorkspace may be given multiple references to allow concrete and visible experimentation with reference semantics, and to emphasize the fact that a reference can be a many-to-one relationship. To ensure that object destruction is interactively visualised, if the sole variable holding an object is assigned null in the OUWorkspace the object will be visibly garbage collected and the graphical representation of the object will disappear from the graphical window. Further concepts, such as refactoring, interfaces (which are taught very early), broadcast dependency and simple coding patterns are explored in similar ways.

5. Coding style

Of course, students move on from writing snippets of code in the OUWorkspace to modifying methods of existing classes before moving on to develop classes of their own. In writing code we enforce a verbose coding style that reinforces object ideas. We insist that within methods an object's own instance variables are always qualified by `this` and that class variables are always qualified by the class name – we do this because we want to make clear the distinction between object and class and also to avoid any confusion with similarly named class or local variables. Similarly messages within a method to the object executing that method are always qualified by `this` (or of course `super`); to miss out the qualifier is to make the message-send look like a procedure call and we wish to reinforce that most of the processing in an OO program involves sending messages to objects. Note, in the context of objects we always talk in terms of sending messages to objects, *not* invoking methods. Messages are polymorphic, methods are not; the decision on which method to invoke is not determined at compile time but at run time by the JVM depending

on the class of the object. However with static (class) methods we do talk in term of method invocation as method resolution can be determined at compile time. Instance variables are invariably made private to enforce data hiding and where necessary public accessor methods are written.

6. Evidence of the effects of the approach

The primary aim of this paper has been to describe and analyse a teaching approach and its systematic basis in a set of principles. It is not primarily about an empirical examination of the effects. However, there are some sources of evidence available that have some general bearing on the effects of the teaching approach on students and teachers, which we will now consider.

The first source of evidence comes from the routine student surveys that the Open University carries out for all courses. These surveys present the opportunity to compare students' general opinions of M255 with a pre-existing course that took a far more conventional approach to teaching Java. More specifically, prior to M255 the only 2nd level course to teach Java was the 20pt course M254, which had four presentations between 2004 and 2006. This course was traditional in its approach, for example, starting off with *main()* to print a string to the standard output, teaching loops and iteration before addressing objects. The students on both M255 and M254 were surveyed in the autumn of 2006 by the University's Institute of Educational Technology as part of a survey of all our faculty's courses. In the survey students were asked to rate their extent of agreement to a number of statements (table 1). The results are indirectly relevant to our claims in that they afforded an opportunity to refute or weaken the claim that our approach is beneficial to students.

		M254	M255
The course was more difficult than I expected.	Definitely or mostly agree	59.1%	20.5%
The course met my expectations.	Definitely or mostly agree	79.4%	89.1%
Overall I was satisfied with the teaching materials provided on this course. (For example printed text; CD ROMs; DVDs; online materials.)	Definitely or mostly agree	80.4%	90.4%
I enjoyed studying this course.	Definitely or mostly agree	82.2%	87.2%
I would recommend this course to other students.	Definitely or mostly agree	72.9%	85.7%
The course met its stated learning outcomes.	Definitely or mostly agree	84.1%	89.6%
The course provided good value for money.	Definitely or mostly agree	67.0%	78.3%
Overall I am satisfied with my study experience.	Definitely or mostly agree	79.4%	90.4%
Overall I am satisfied with the quality of this course.	Definitely or mostly agree	79.4%	89.1%

Table 1

The simplest relevant observation from this data is that for all nine questions, students expressed more positive opinions about M255 in comparison to the more conventional M254.

Another source of feedback comes from the Open University's Course Reviews web site where students are encouraged to comment on any course they have studied (http://www3.open.ac.uk/coursereviews/). Two students commented specifically on the object-oriented nature of the course, as follows.

> *"A very enjoyable course. I have done some programming before, but had never really got my head around Object-Oriented Programming - until this course. The course content kept me interested and explained everything ever so clearly. I'm now really looking forward to, and am confident about, studying the higher level courses in this area."*

“This was a truly excellent course that really does get you started in OO programming and Java. It does exactly what it says on the tin and actually helped me a great deal in moving to a new job where I am programming in C+(a similar language to Java.) The course materials were great and the software equally good (apart from a few bugs in the OUWorkspace which will hopefully be ironed out in future presentations.) 10/10 "

A third indirect source of evidence about the effects of M255’s approach is the figures for success on the course compared with its more conventional predecessor (tables 2 & 3).

M255 (Oct '06)	HEFC Return	Percentage of students included in HEFC returns who sat the exam
Total	1409	63
New students	131	63
Continuing students	1278	63

Table 2

M254 (Oct '06)	HEFC Return	Percentage of students included in HEFC returns who sat the exam
Total	309	60
New students	16	56
Continuing students	293	60

Table 3

Perhaps the most interesting observation here is that the retention of new students was significantly increased, while more generally, retention was up slightly. Evidence of this kind bears only obliquely on our assertions, however, again it did at least afford an opportunity to rebut our claims.

The fourth source of evidence we shall consider comes from a small opportunistic poll of tutors who had taught on both courses (table 4). The sample is opportunistic in that all nineteen tutors were polled, but only some were able to respond in the limited time available. The sample is not statistically significant (six tutors), although extremely similar results were obtained from slightly larger sample of eight tutors on the course, by including responses from two of the authors of this paper. However, we will limit our comments here to the responses of the group uninvolved in this paper.

Tutors at the Open University are frequently asked their opinions about courses they teach, and the institutional culture is such that critical opinions are freely and routinely given. The questionnaire covered three of the most salient features of the course, and considered ten aspects of each of these features. Tutors were asked to respond on a five-point Lickert scale as follows: Definitely agree = 2, Mostly agree = 1, Neither agree nor disagree = 0, Mostly disagree = -1 and Definitely disagree = -2 . The results of the questionnaire are shown below (table 4). Entries in the table indicate the proportion of the six tutors mostly agreeing or agreeing strongly with the statements as applied to the different features of the course. Some key observations are that the sample were *unanimous* that all three selected features of the course benefited students. Interestingly, there was less unanimity about benefits to tutors. However, it is worth noting (not shown in the table) that *none* of the sample of tutors mostly disagreed or definitely disagreed with *any* of the statements about any of the features. In other words, the least positive opinions expressed in response to any question were neutral – there were no negative responses to any question. However, when the sample was expanded to eight course teachers (not shown in the table) by including the authors, some negative opinions were recorded. This was due to the fact that one author considered one aspect of the object oriented programming style (a stress on accessing instance variables via accessor methods, rather than directly) to add one more element of verbosity to an already relatively verbose programming language. However in all other respects, results from the slightly larger group were very similar.

	Interaction with objects via memorable microworlds in M255	The use of the OUWorkspace in M255 to interact with live objects	The explicitly object-oriented programming style of M255
Benefits students	6/6	6/6	6/6
Benefits tutors	3/6	4/6	5/6
Helps students to visualise object concepts	6/6	6/6	6/6
Helps students to grasp object concepts quickly	6/6	6/6	6/6
Helps students to focus on object fundamentals rather than syntactic detail	6/6	5/6	5/6
Helps students to form a clear conceptual model	6/6	5/6	6/6
Helps students to remember object fundamentals	6/6	5/6	5/6
Helps students to explore the syntax and semantics of Java	4/6	4/6	3/6
Makes the course more interesting	6/6	5/6	4/6
Makes the course more fun	6/6	3/6	2/6

Table 4

Tutors were also given the opportunity to contribute free form comments on any issues raised by the questionnaire. Principal issues were raised as follows.
Several tutors commented on a specific technical limitation of the OUWorkspace (it is currently unable to deal with generic collections as introduced in Java 1.5), which means that it cannot be used directly to manipulate such collections. Some tutors commented on the usefulness of the OUWorkspace to tutors as well as to students.

"The OUWorkspace is wonderful – I found it very useful when writing my own code and in preparing examples (although there are things that you can't do with it)."
"The workspace used an old version of the JDK so not all Java syntax could be explored interactively which was frustrating for student/tutor. Otherwise, it was an excellent course, taking and adapting the first half of M206."

ISSN: 1473-7507

One tutor commented on neglected opportunities.

"I do think it would have been useful to use the BlueJ facility that lets you create an object and send messages to it by clicking on its representation in the BlueJ desktop. I think it better connects classes and objects than the OU workspace. I also think it would help to teach them to use the interactive debugger."

Some tutors noted the benefits to students and tutors of interweaving early coding with memorable microworlds, and the extent to which this encouraged confidence.

"It allows me as tutor at tutorials to talk in more concrete terms."

"My only other comparison with another OU course in Java is M257 [a follow on Java course], but for the initial hands on approach, the model borrowed from M206 seems to allow progress at an early stage to confidence that is crucial to good success. This appears to be true for both experienced students and those just starting."

This view of the microworlds was not universal.

"My only concern is with the microworlds which some students (and tutors) find irritating. I don't have this view - I think they are very helpful."

Some comments concerned the explicit object-oriented style of coding in M255.

"The OO style isn't just preferable, it's essential! – although students who then go on to M257 [a follow on Java course] seem to get upset that they aren't required to stick to the same rules there. Again, I don't have a problem with this – we don't live in an ideal world, and the sooner they get used to having to do things differently on different occasions, the better."

Some tutors commented on the course's foregrounding of object-oriented concepts over syntactic detail.

"I think the course does a good job in abstracting the essential concepts of OOP before they get bogged down in the complex syntax and semantics. The rapid progress of M257 students is a good sign that we are getting it right."

Each category of evidence that we have considered is only weakly indicative in terms of strict relevance to our claims. Still, each category did at least offer an opportunity to rebut

our claims, and in each case, to the limited extent that the evidence is able to afford relevant support, relatively clear support was given.

6. Conclusion

In terms of the goals that we set ourselves for the course, namely to teach object concepts through the vehicle of Java while approaching as closely as possible the clarity with which we were able to teach them using a pure object language, we believe we have had a reasonable degree of success, but it is open to more rigorous empirical evaluation to determine exactly to what degree, and in what respects, we have been successful.

The need to deal with the large number of irregularities, inconsistencies and special cases in Java curtailed the breadth of detail we were able to cover compared with the previous course using a pure object language (M206). For example in M206, students with no previous experience of programming gained firm grasp not only of constructing and modifying MVC user interfaces, but also extensive detail of the separable interface architecture and the mechanisms used, as well as quite complex forms of object-oriented iteration (Griffiths et al, 1999).

We believe that teaching fundamental object concepts lucidly, over and above the teaching of skills in particular programming languages, is not an optional goal – it is vital. We recommend consideration of the strategies outlined in this paper to teach object concepts effectively, whatever language is used as a vehicle.

References

Bates, R. (2004) *Opinion: Schizoid Classes*, ACM Queue: Tomorrow's Computing Today, 2(6), pp. 12-15.

Griffiths R., Holland S., Woodman M., Macgregor M., Robinson H. (1999), *Separable UI Architectures in Teaching Object Technology* In: Proceedings of the 30th International Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, pp. 290-299.

Holland S., Griffiths R., Woodman M. (1997) *Avoiding Object Misconceptions* In: Miller J., E. (Ed.), Proceedings of the 28th ACM Special Interest Group on Computer Science Education Technical Symposium on Computer Science Education, ACM Press, pp 131-134.

Kay, A., C. (1993) *"The Early History of Smalltalk"*, ACM SIGPLAN Notices, 28 (3), pp. 69-95.

Kölling, M., Quig, B., Patterson, A., Rosenberg, J. (2003), *The BlueJ system and its pedagogy*, Journal of Computer Science Education 13 (4), pp. 249-268.

Mortensen, S. E. (2001) *Why Java Isn't Smalltalk: An Aesthetic Observation*, Smalltalk Chronicles, 3 (1).

Woodman M., Griffiths R., Macgregor M., Holland S. (1999), *OU LearningWorks: A Customizable Learning Environment For Smalltalk Modules* In: Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society Press, pp. 638-641

Woodman M., Griffiths R., Robinson H., Holland S (1998) *An Object-oriented Approach to Computing*. In: Haungs J. (Ed.), Proceedings of the ACM Conference on Object-oriented Programming, Systems and Languages, Educators' Symposium Addendum, ACM Press.

ISSN: 1473-7507