Avoiding Object Misconceptions

Simon Holland, Robert Griffiths, Mark Woodman
The Open University
Faculty of Mathematics and Computing
Walton Hall, Milton Keynes, MK76AA, United Kingdom

s.holland@open.ac.uk
r.w.griffiths@open.ac.uk
m.woodman@open.ac.uk

Abstract

This paper identifies and describes a number of misconceptions observed in students learning about object technology. It identifies simple, concrete, measures course designers and teachers can take to avoid these misconceptions arising. The context for this work centres on an introductory undergraduate course and a postgraduate course. Both these courses are taught by distance education. These courses both use Smalltalk as an introduction to object technology. More particularly, the undergraduate course uses Smalltalk as a first programming language.

Distance education can limit the amount and speed of individual feedback that can be given in the early stages of learning. For this reason, particular attention has been paid to characterizing measures for avoiding elementary misconceptions seen in beginning learners. At the same time we also address some misconceptions observed in postgraduate students. The pedagogical issues discussed are of particular importance when devising an extended series of examples for teaching or assessment, or when designing a visual microworld to be used for teaching purposes.

Introduction

Object concepts are often taught, especially in the first few lessons, with a great deal of practical demonstration during lectures, and with a lot of expert help on hand for lab work. This is not because object concepts are intrinsically difficult, but because the subject does offer many opportunities, especially in the early stages, for students to develop misconceptions, which can be hard to shift later. Such misconceptions can act as barriers through which later all teaching on the subject may be inadvertently filtered and distorted.

SIGCSE '97 CA, USA

If teachers have sufficient preparation time, if there are adequate practical demonstrations, and if expert advice of sufficient quality is on tap during practical sessions, these problems can often be avoided. However, teaching conditions are not always ideal; there is not always sufficient practical help of an adequate kind; and even in the best circumstances, some students are likely to acquire some misconceptions.

The problem of avoiding object concept misconceptions can be potentially particularly acute in the case of distance education. In this context it is often impractical to give frequent demonstrations or to provide immediate feedback to student queries during such demonstrations. The problem is made more acute when the student population necessarily includes a mixture of arts students with no programming experience, and computing students with previous experience of a procedural programming language.

For these reasons we are identifying misconceptions in the learning of basic object concepts. Such knowledge could be used by teachers writing or reviewing teaching material to ensure that the choice of examples, problems, terminology, and teaching sequence does not inadvertently foster common object misconceptions. This work has arisen while developing two Open University courses on object technology, the postgraduate course *Object-oriented Software Technology (M868)*, and the undergraduate introductory computing course *Computing: an Object-oriented approach (M206)*.

Although this work was carried out in the context of a distance education course, we believe it is applicable whatever teaching style or medium is used. In this work we concentrate almost entirely on the elementary, early parts of the curriculum. We believe, in the light of our experience of using some of these patterns in postgraduate teaching, that it is equally important to get these basic points right in postgraduate or more advanced courses.

Avoiding object/variable conflation

Many early teaching examples feature classes with a single instance variable. There is a danger that some students with previous experience of procedural programming may generalize prematurely from these examples to develop the misconception that objects are in some sense mere wrappers for variables. It is trivially easy to avoid this misconception by the simple discipline of ensuring that all introductory object examples make prominent use of classes with more than one instance variable. For example, the classic bank account example can be very useful, but there are dangers in introductory teaching in using an *Account* class that is limited to a sole instance variable *balance*. There should be at least two instance variables to avoid the "object as a kind of variable" misconception.

However, many early teaching examples also feature classes in which all instance variables are expected to hold objects of the same class. For example an introductory *Account* class in the early stages should not be limited to instance variables that hold objects which are all number objects, e.g. *balance* and *limit*. Some students may be influenced by such examples to develop the misconception that instance variables of objects of a given class must all refer to objects of a single class. Therefore as a remedial measure, classes in early teaching examples should have at least two instance variables, which expect objects of different classes. An introductory account class, for example, could have two instance variables, e.g. *balance* and *name*. Care should be taken that the use of an instance variable such as *name* does not lead to confusion between object identity and object attributes—which we deal with below.

Objects are not simple records

Many students, and indeed some instructors creating teaching and assessment examples focus on examples where an object behaves essentially like a database record, or repository for inert data. A case in point might be a music CD class, in which each object represents a music CD, and stores information on the title, artist, tracks, etc. This overemphasizes the data aspect of objects at the expense of the behavioral aspect. The practical danger is that students may come to tacitly assume that all objects are simple, inert records. They may fail to realize that the behavior of some objects may alter substantially depending on their state. This misconception can be avoided by using introductory object examples that prominently feature classes where the response to a message is substantially altered depending on the state of the object. A simple example object whose behavior is affected by its state might be an *Account* object that refuses a debit request when an overdraft limit is reached. Debit requests are not accepted until the limit is changed, or until more money is credited.

Work in methods is not all done by assignment

The kind of code that students see in the first methods they look at can be very influential on their thinking. This is particularly true when the course is an introduction to programming. For example, in many introductory teaching examples, using an *Account* object, the instance variables recording balance, etc., refer to immutable objects such as numbers. For this reason, the *Account* methods that manipulate such instance variables tend to use assignment rather than method passing.

As a piece of programming, of course, and as a single teaching example, there is nothing in the least wrong with this. However, there is a danger that exclusive exposure to this way of changing state can foster the impression that work in methods is exclusively done by assignment (and not by message passing). If early teaching examples happen to be chosen so that all state is represented by immutable objects, such as number objects, it is hard to avoid this danger.

We have observed that even very experienced students pick up this impression from such examples and that this misconception can lead to an over reliance on assignment and a procedural style of coding. To avoid the problem one should use examples where the values of instance variables are not invariably immutable objects. A simple example teaching domain that avoids this problem might involve a business that buys and sells various products which themselves have state.

Object/class conflation

When presenting a series of examples in the early stages, it is easy to find oneself using examples in which only a single instance of each class is used. At some stage or another, some students tend to become confused between classes and their instances. Indeed, in some object-oriented languages there is no distinction. So as to not foster this misconception, it is good practice to always work with several instances of each class in any given teaching example.

Identity/attribute confusion

In the traditional bank account example, frequently just two instance variables are used, *name* and *balance*. This is admirable in that there are at least two instance variables, that are not of the same type, and the example is intuitively clear and familiar to most people. However, in the minds of students with previous exposure to database concepts, the *name* instance variable in this example (whatever that variable is called) can give rise to anxiety and misconceptions. There is a tendency to confuse the *name* instance variable with the identity of the object, or with a variable that refers to the object (e.g. *myAccount*). These confusions can lead to further misconceptions, some of

which are itemized below:

- only one variable can reference to a given object at a given time;
- once a variable references a given object, it will always reference that object;
- a variable that refers to an object uniquely specifies it for all time;
- if you have two different variables, they must refer to two different objects;
- you can ask an object what variables refer to it;
- two objects of the same class with the same state are the same object;
- two objects with the same value for the *name* attribute are the same object.

Rather than try to deal with these misconceptions by arguing or talking about them, the easiest approach is to immediately let the students experiment with a set of counter examples (no pun intended). These counter examples can be summarized as follows:

Multiple assignments: get students to assign a single object to three variables at once. Demonstrate that each variable references the same object by showing that state changes effected via any one reference can be inspected immediately via all of the other variables.

Re-assignment: get students to assign a different object (ideally of an altogether different class) to one of the variables, and then show by sending messages and inspecting the result that the variable now refers to a different object, whilst the other variables still refer to the original object.

Swapping: swap the variables that refer to two objects, using an intermediate holding variable.

Instance variables with the same value: show that two demonstrably different instances may have the same value for the same instance variable.

Objects with identical state: prove that two instances with identical state are not the same object by sending messages that make their states diverge.

Conflation of textual representation of objects and references to objects

One of the earliest expressions evaluated by many students using *show it* is typically something like.

```
2 + 2
```

Students are told that every message returns a message reply object, and students see the textual representation `4` on the screen. It is natural (and correct!) to identify the textual representation `4` on the screen as a reference to the message reply object. On the other hand when an expression such as:

```
myAccount:= Account new
```

is evaluated using *show it*, students see a textual representation such as `anAccount`. In this case, it is natural (and incorrect) to identify the textual representation `anAccount` as a reference to the message reply object. If this confusion is acquired and not addressed, it can be easy for the student to conflate a reference to an object with its printString. (This confusion has been seen in assorted postgraduates, and programmers coming from a LISP or functional programming background.)

The cleanest way to defuse this misconception is to teach reference as a first class concept of equal important to the concepts of object, message and class. In particular, the concept of variable is treated as a special case of the more general concept of reference. Other examples of valid references are number, string, character and array literals and message expressions (i.e. message replies). In concrete terms, students are asked to cut and paste textual representation of message replies for various classes, and to evaluate expressions that treat these textual representations as references to objects. Based on the results, they are asked to decide whether, in general, textual representations produced by *show it* are valid references to objects. They are also asked to give example classes for which the textual representation produced by *show it* are valid references to objects.

Limitations

The misconceptions and practices described here are ones that have been seen in tutoring postgraduates, or during developmental testing of the undergraduate course (M206) and its associated CD ROM. Some of the measures to avoid the misconceptions have been tried out on postgraduate students, and others on developmental testers for the undergraduate course. Clearly no claim is made of formally provable links between bad practices and corresponding misconceptions, nor between suggested measures and avoidance of misconceptions. Links are based on personal experience of teaching and broad support from the results of developmental tests. All of these misconceptions, and the measures to avoid them could probably be usefully treated as pedagogical patterns, in contrast to the discursive treatment given here.

Conclusions

This paper has identified and characterized several misconceptions observed in students learning about object concepts, and has described simple teaching measures to avoid them. Pedagogical issues have been discussed of particular importance when constructing teaching or assessment examples.

The paper presents six out of a larger number of misconceptions that we have investigated from our work on the undergraduate and postgraduate courses dealing with object technology. Two useful extensions of the work, would be to characterize more misconceptions and measures, and to recast them as patterns.

References

1. Beck, K. *Smalltalk Best Practice Patterns Volume 1: Coding* (pre-publication draft) First Class Software Inc., Boulder Creek, CA., 1996.

2. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*, Addison Wesley, New York, 1995.

3. Prieto, Maximo. The importance of learning Object-oriented thinking, *Proceedings of Workshop on Learning, Training and Teaching in Object Technology*, part of European Conference on Object Oriented Programming, Aarhus, Denmark, 1995.

4. Leonardi, C., Prieto, M., Rossi, G., Levato, A., Echarri, F., Maciel, R. Micro-worlds: A tool for learning object-oriented modeling and problem solving. *Proceedings of Educators Symposium, OOPSLA '94*, Portland Oregon (October 1994).

5. Woodman, M and Holland, S. From software user to software author: an initial pedagogy for introductory object-oriented computing. *Proceedings SIGCSE/SIGCUE '96*, Barcelona , Spain (June1996).