Employing Object Technology to Expose Fundamental Object Concepts

Mark Woodman, Rob Griffiths, Malcolm Macgregor, Hugh Robinson, Simon Holland,
Computing Department, The Open University, Walton Hall,
Milton Keynes, England MK7 6AA
tel: +44 1908 274066
email: m.woodman, r.w.griffiths, m.d.macgregor, h.m.robinson, s.holland@open.ac.uk

Abstract

This paper reports on the utilisation of object technology in a university-level course on software development, specifically designed for distance learning, and now enrolling over 5,000 students per year (average age 37) in the UK, Western Europe and Singapore. M206 'Computing: An Object-oriented Approach' embodies a practical, industry-oriented view of computing, which has resulted in a prestigious Award for IT from the British Computer Society. The course is large, representing one sixth of a degree, and the full panoply of educational media is used in its delivery. M206 introduces computing via an object-oriented approach. To ensure that the concepts involved in the development of complex software systems are well grounded and applicable by neophytes, we have employed object technology itself to realise our pedagogy, investing over 50 person years of academic effort. The paper describes the agenda for the course, its object-oriented pedagogy and our strategy for delivering it. In particular, we explain our object systems approach, how we avoid misconceptions about objects, our analysis and design method, and the Smalltalk programming environment we have developed to support our pedagogy. The environment is crucial to our strategy and exemplifies our strategy of pedagogic utilisation of object technology: hence, the paper details one of our object-oriented simulations and outlines how our strict adherence to the separation of view and domain model leads to technical innovations. Concluding remarks reflect on the benefits a reflexive strategy can bring especially in high-pressure industry training situations.

1 INTRODUCTION

The Open University (OU) is the UK's largest university; its primary mission is to make higher education available to adults regardless of their personal circumstances and earlier educational achievements and and it courses are offered almost exclusively in the distance mode. Since it was established in 1969 more than two million people have studied with the OU the UK, Europe and world-wide. After fifty person years development, in 1998 the OU launched its flagship course M206 *Computing: An Object-oriented Approach* – a radical introduction to designing and writing complex software systems. It is a large course – 440 hours of study during 33 weeks, worth one sixth of a degree. It goes beyond what any organisation has attempted by providing to ordinary *users* of computer systems the resources to become *developers* of them and by teaching them object technology from the start. M206 is enrolling over 5,000 students per year; typically they are aged 37 and in middle management, the social and educational impact of tens of thousands first learning to think of software in object-oriented terms will be both dramatic and immediate. From early 1999, an additional 600 are taking the course at the Singapore Institute of Management, and it will soon become available in the USA.

When devising the syllabus for this course in the early 1990s, we had to assess what we believed to be the technologies for software development relevant to the end of the decade. To this end we reviewed and refined our plans in industry as well as academic fora. From industry it became clear that they needed people who could think in terms of complex, long-running software systems, not just in terms of simple input–process–output programs. One extremely effective view of complex systems is embodied by object technology in which systems are considered to be composed of parts which perform the computational work of the system by sending each other messages. We also believed that the Internet and the infant World-wide Web

would be of huge relevance. Therefore, object and network technologies, are central to the course and essentially define its approach to software development. After much debate [1] we decided to adopt an 'objects-first' approach to computing and that the programming language which would best suit our agenda was Smalltalk-80 [2, 3], an industrial-strength language and arguably the first really practical object-oriented programming language. For us its primary benefits were that the language is based on just a few concepts and its programming environments have the following properties:

- they are simple embodiments of object technology;
- they lend themselves to tailoring;
- they are suitable environments in which to produce of student-alterable simulations.

M206 is not just about object-oriented programming, but also, analysis and design, networks, operating systems and human–computer interaction (Details of the syllabus and multimedia presentation can be found at www-cs.open.ac.uk/~m206.) In addition, the course emphasises the human dimension in software processes, and that it is people and how they deal with complexity that often determines success or failure. To reflect this, group working via an electronic conferencing system takes place throughout the course. Hence, the course was designed to be broad as well as deep, and its size (i.e. number of study hours) provided us with the means to take a holistic approach to computing. In other words we did not have to break our account of computing into perceivably discrete pieces, but could offer an account encompassing most of the aspects involved in real systems. The syllabus topics for the course are:

- Systems Thinking
- Network Computing (the Internet, Web, conferencing)
- Human–Computer Interaction
- Object Technology (and programming with Smalltalk)
- Exploratory Programming Environment (LearningWorks)
- Group Working
- Software Development Processes
- Modelling for Object Analysis and Design
- Practical Computing (e.g. operating and database systems)

On completion of the course, it was our goal that a student, among other things would:

(i) have a extensive understanding and vocabulary of computing, software and object technology;

(ii) have sufficient knowledge of the object-oriented paradigm to analyse artefacts and problems in terms of it, to design system parts, and to complete or extend applications;

(iii) be capable of developing applications including appropriate graphical user interfaces;

(iv) be able to discuss the issues involved in large scale software development and group working and be able to engage in such processes;

(v) be able to describe, analyse and implement user interfaces;

(vi) be able to contribute to a CRC-style of object-oriented analysis.

As may be inferred from the above topics and learning outcomes, the course is very practical, and a learning-by-doing pedagogy was designed accordingly [4] and in a manner that would be effective in the distance mode. We needed to focus both on the syllabus, especially the concepts of object technology, and what educational technologies we could deploy. Here we decided on a fully multimedia approach: the distance learning materials include some fifty illustrated text documents (around thirty pages each), associated software, a Web site, 12 nationally broadcast television programmes produced with the BBC, interactive CD-ROM [5], the Smalltalk programming environment, communications software, and computer conferences [6]. Crucially, we experimented with prototype materials and with different media mixes,

eventually settling on a consistent design [7] within which our LearningWorks programming environment was core.

In subsequent sections we describe the initial pedagogy for the course, which specifically addressed common misconceptions which he had identified, we describe our analysis and design method, our use of object-oriented simulations, our consistent separation of domain model and user interface, and the ways in which we used especially LearningWorks to support our pedagogy.

2 INITIAL PEDAGOGY

The course begins with elementary computing and software vocabulary followed by an exploration of two commonplace applications that we have implemented – a word processor and a drawing application. While we teach their use for practical purposes later in the course (they may be used for writing assignments), the main pedagogic use is to introduce object concepts. For example, we introduce the notions of object *state* and state-related *behaviour* by discussing what is going on when a word processor document is sent a *close* message: if nothing has been typed into the document, it disappears quietly; if its state has been changed by typing it reports this and the word processor presents a dialogue box asking for confirmation that the document contents is to be discarded.

Similarly, we explain the way in which a drawing application changes some text to italics can be modelled by a set of interacting objects sending messages to each other. In this way users, are encouraged to think about appropriate models of the behaviour they are use to. These ‘appropriate’ models should lead seamlessly to object-oriented programming.

Separation of concerns is an important principle of software development that is clearly crucial in well-designed object programs. To be able to separate consideration of the user interface from the functionality of software we next introduce human–computer interaction. As part of the transition from user to software developer students learn about the important and unimportant similarities and differences of different GUI operating systems. They are taught that names (e.g. of buttons) and icons can only be suggestive – in other words the difference between syntax and semantics.. They learn early about conceptual models, about affordance, metaphors and about bad design of user interfaces.

Having introduced object ideas in an intentionally vague and gentle fashion, we next progress to an amphibian microworld of frogs and toads, which is discussed later. To avoid students thinking that programming is all about frogs, toads, and the like, we of course do introduce more mundane computer science examples. However central to our pedagogy and to the design of LearningWorks simulations and tools is the notion of *progressive disclosure*: early on, very few classes are made visible, and within these very little implementation code is not made visible, but as the course progresses, we provide increasingly more powerful browsers that eventually reveal the whole Smalltalk library. Indeed. once we have dealt with the basic concepts of object-oriented programming and Smalltalk (and incidentally have taught students how to use e-mail, conferencing and the World Wide Web) we can progress to more elaborate visualization tools which can be used to explore more Smalltalk concepts: expression series, message answers, nested expressions, reference, variables, etc.

In the next section we detail how our initial Smalltalk examples have been chosen to avoid common object misconceptions which we have observed in students.

3 AVOIDING OBJECT MISCONCEPTIONS

In face-to-face instruction, object concepts are often introduced 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 correct later. Such misconceptions can act as barriers through which all later teaching on the subject may be inadvertently filtered and distorted. The problem of avoiding object concept misconceptions can be 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 includes a mixture students with no programming experience, and those with previous experience of a procedural programming language. For this reason, we have paid particular attention to characterising measures for avoiding elementary misconceptions seen in learners who are either new to computing or who have some experience with procedural programming. The misconceptions we concentrated on were ones we identified during the developmental testing of this course with prototype materials [7].

Often, early teaching examples feature classes with a single instance variable; we found that 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. We have avoided this misconception by the simple discipline of ensuring that all introductory object examples make prominent use of classes with more than one instance variable. Furthermore some students 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 have at least two instance variables, which reference objects of different classes.

We have also avoided 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 overemphasises the data aspect of objects at the expense of the behavioural 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 behaviour 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 behaviour is affected by its state might be an `Account` object that refuses a debit request when an overdraft limit is reached. Such debit requests are not accepted until the limit is changed, or until more money is credited.

The kind of code that students see in the first methods they look at can be very influential on their thinking. For example, in many introductory teaching examples, instance variables refer to immutable objects such as numbers. For this reason, methods that manipulate such instance variables tend to use assignment rather than method sending. 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 tends to foster the impression that work in methods is exclusively done by assignment (and not by message sending). 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 we have used examples where the values of instance variables are not invariably immutable objects, but instead objects which themselves have state.

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, we have ensured that examples use several instances of each class in any given teaching example.

In the traditional bank account example (which we ourselves have used), 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 itemised below:

- only one variable can reference a given object at a given time;
- once a variable references a given object, it will always reference that object;
- 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 avoidance examples summarised 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.

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

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

Hence to clearly teach fundamental object concepts we had to develop appropriate materials, programming environment and simulations to practice such activities.

4 ANALYSIS AND DESIGN

Our approach to analysis and design is loosely centred around the CRC approach of Wirfs-Brock *et al*. [8] but with a flavour of the more formal treatment of associations given by Cook and Daniels [9]. This results in an overall 'feel' for students that is both sufficiently informal and intuitively attractive but nevertheless has sufficient rigour to underpin (and reinforce) the fundamental object concepts. Analysis proceeds with the identification of classes, associations, relationships and invariants, with understanding being expressed via the incremental development of an object model that consists of both a graphical and a textual representation, the two complementing one another. Importantly (and perhaps in contrast with the informality of the original CRC approach) all features of the model are articulated in terms of classes (and associations) and instances. For instance, much of our early teaching in this area is done via an example involving a hospital with wards, doctor, patients and nurses with the statement that some nurses are designated to supervise one or more other nurses *on the same ward.* This is first expressed as the rule that a nurse who supervises other nurses must be assigned to the same ward as the nurses he/she supervises. This rule (which, in fact, is initially expressed in a much less formal fashion) is then articulated in terms of instances as follows:

```
Invariant
Any given instance of Nurse is associated with other instances of Nurse via the
Supervises association. Each of these other instances of Nurse must be associated, via
Is-Staffed-By, the same instance of Ward that the given instance of Nurse is associated
```

`with via Is-Staffed-By.`

Indeed, this insistence on the articulation of understanding in terms of object model constructs is a potent and reflexive tool, where issues that arise in some statement of requirements are articulated as object model constructs, thereby giving rise to questions that need to re-articulated back to the (user) requirements and their solutions re-expressed back again in terms of object model constructs. As we emphasise in one of the analysis and design chapters:

> This need to clarify the analyst's understanding of the detailed meaning of the application area should not be seen as some deficiency in object-oriented analysis. Rather, it is one of the great strengths of an approach such as that based on classes, associations, responsibilities and collaborations that the activity of constructing an object model forces questions about the meaning of the application area to be asked. It is also indicative of a characteristic of good analysis – it is essentially an active exploration of meaning rather a passive representation of 'requirements'.

Such an approach is followed through in later stages of analysis (and design), when examining the necessary collaborations and the assignment of responsibilities – students are forced (sometimes, painfully) to express understanding in terms of instances, how references to such instances may be obtained (and the business of a navigation path) & how instances may fulfil their responsibilities. This approach is underpinned with a number of key pedagogical (and, indeed, epistemological) characteristics which include:

1. The acquisition and practise of dispositional skills in the identification of classes, associations, responsibilities and collaborations by exposing students to a range of problem scenarios where those skills may be deployed.
2. The separation of concerns (user interface versus problem domain) as crucial in the successful design of systems.
3. The importance of re-use within design.
4. An emphasis on the importance of producing credible and recognisable (for the student) Smalltalk code as the demonstrable product of the analysis and design process.

The overall outcome of this learning process is a practical competence in – and an understanding of the purpose and nature of – the activities that must be engaged in when carrying out object-oriented analysis and design, rather than an exploration of the nuances of some prescriptive method. Importantly, our emphasis on fundamental object concepts is not some sterile mouthing of paradigmatic fundamentalism: rather it is an emphasis that allows us to actively explore and comprehend the reality of a world that is suffused with the full complexity and richness of human life.

5 OU LEARNINGWORKS

A core course component heavily used in teaching object concepts is OU LearningWorks, a Smalltalk environment which we developed [10] from work with Adele Goldberg and her colleagues on the LearningWorks [11] programming framework [12]. After an impartial analysis of how students used a prototype of our LearningWorks environment [7], we adopted a consistent design for organising the teaching materials, simulations and programming tools into plug-in Smalltalk modules called LearningBooks. As far as learners are concerned LearningWorks offers a set of LearningBooks whose user interfaces follow a book metaphor: a LearningBook is organised into sections, and sections into pages. Each page is in fact an application which can be a Web-style HTML page, a programming tool, or a simulation. And, like in a loose-leaf binder, a page may be 'detached' from its book and left conveniently on the desktop, allowing the user to continue to view a page from one section, having moved to another. In our standard LearningBook design, the first section of a LearningBook always contains an HTML-browser page of practical activities and discussions, a glossary of terms, and a page for taking notes (see Figure 1). The second and subsequent sections of each

LearningBook contain Smalltalk classes, programming tools such as class browsers and workspaces (pages for evaluating Smalltalk code) and microworld visual simulations which students use to carry out the practical activities.

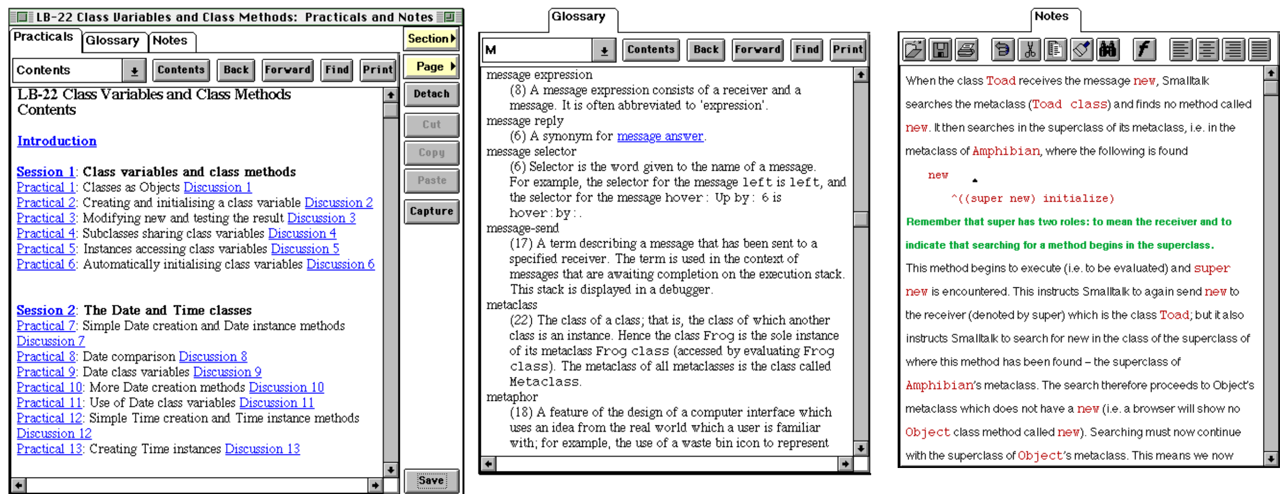


Figure 1

An overriding requirement of the environment was that it should initially constrain novices and hide detail from them but would progressively disclose the facilities and rich detail that would be familiar to the experienced practitioner, this was especially important due to the complexity of object-oriented class libraries. This powerful mechanism amongst other things allows a LearningBook author to:

- define which classes are visible to the debugger and the class browser;
- define for each visible class those methods whose code can be viewed and edited by the student;
- define for each visible class those methods whose code cannot be viewed by the student,

Hence as the course progresses, the LearningBooks can provide increasingly more complex class browsers which expose more of the complexity of the Smalltalk library.

Next we describe a microworld which was designed to be a touchstone for students and teachers alike – a simulation with which they become intimate and using which they can reason about *all* object concepts taught in the course.

6 SIMULATIONS WITH OBJECTS

Smalltalk is inextricably linked with simulation [13]. Indeed, an object-oriented system is often described as a simulation, or model, of some part of the real world or a business enterprise. We were therefore culturally well disposed to introduce simulations into our interactive learning environment in the form of microworlds. One in particular was used as a touchstone for reasoning about object concepts – an amphibian microworld that models the behaviour of instances of classes `Frog` and `Toad` and of a subclass of `Frog`, `HoverFrog`. The initial simulation given to students is a concrete cartoon-like world consisting of frogs and various other amphibians (two variations are shown in Figure 2 and Figure 3). For the purposes of the simulation, frogs can be made to move their position and change their colour. Via a graphical, highly visual user interface, with usual menus and 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, inspect message replies, and look at how a message to one frog may in some cases cause a frog to send a message to another frog.

The simulation has been devised to expose all object concepts, starting with the simplest: initially the simulation shows objects of the classes `Frog` and `Toad` which a have identical state 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, and brown, and change its position to the 'home' one and move left and right. Students select any of the objects in the microworld from a regular scrolling list (which they much later discover is another object!) and use the GUI widgets to send the corresponding messages. This simple user interface not only allows straightforward message sending to be visualized, it allows apparently advanced notions such as polymorphism to be 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.

Similarly, some menu commands correspond to messages that require arguments. For example, to change the colour of `frog5` to, say red, it must receive a message `colour: Red`; as can be seen in Figure 2 in the graphical user interface we have provided the button menu **colour**, whose menu items include the possible arguments of Blue, Brown, Green, etc.

To expose the notion of inheritance in which a subclass has more state and an extended protocols we have invented a new species of hovering frog, simulated by `HoverFrog` as a subclass of `Frog`. For example, `HoverFrog`s can hover vertically and have height in their state, and respond to messages that its superclass instances cannot, e.g. `Frog`s, do not understand the message `up`. After initial exposure to the simulation, we reveal an addition to the user interface, an input box in which Smalltalk expressions can be entered. This operates in parallel to the graphical part of the user interface. Anything that can be done using the graphical elements can be done using the textual commands. For example, the `Frog` instance `kermit` can be asked to move `left` turn `brown` or turn the same colour as the `Frog` instance `gribbit` by 'evaluating' the following messages

```
kermit left.
kermit brown.
kermit sameColourAs: gribbit.
```

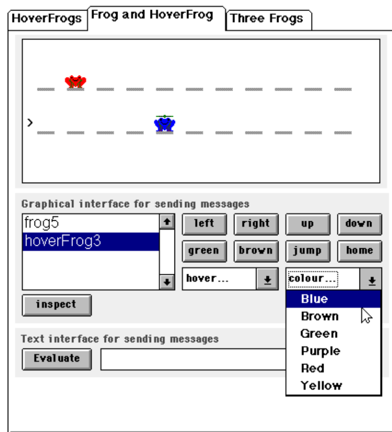


Figure 2

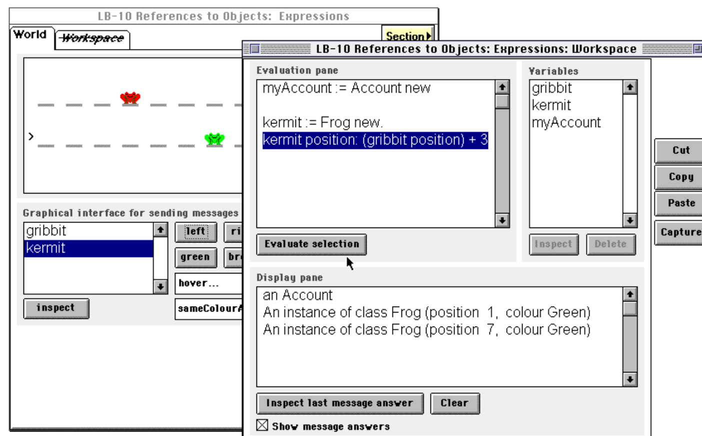


Figure 3

Through the use of this simulation and other simulations, for example, an air traffic control simulation, our students quickly learn the fundamental concepts of object-oriented systems.

7 SEPARABLE USER INTERFACES

Our view of object-oriented systems was consistently supported by adhering to the design principle of separating user interfaces from models. An innovation of LearningWorks we used

heavily to sustain this idea is the provision of *page-local* and *section-local* variables, implemented in page and section dictionaries respectively. In practice the latter are the most useful. For example, any objects of any class created in a Workspace page and assigned to section-local variables are accessible by all pages in that same section.

To facilitate the visualisation of separable user interfaces, assignment, object creation and disposal, our amphibian microworld was built not as an arbitrary application, but, in effect, as a specialised graphical view of the section variable dictionary. This graphical view of the section dictionary is specialised in the sense that that it only displays objects of certain classes of initial interest (e.g. frogs, toads, hoverfrogs). Simply creating an object of the relevant class in a Workspace page and assigning it to a section variable will cause its graphical representation to appear in the amphibian microworld page automatically. A key point is that if the student reassigns variables in the Workspace so that a particular displayed object has no remaining references to it, the automatic garbage collection of that object will be graphically dramatised in its immediate disappearance from the microworld.

Figure 3 shows a LearningBook which is open at the amphibian **World** page. Next to it is a **Workspace** page that has been 'detached' from the same section of the book, i.e. placed on the desktop. Students can send messages to amphibian objects either by clicking buttons on the amphibian **World** page, or by sending messages to them by textually sending messages to amphibian objects in the **Workspace** page. In either case any state-changing messages such as `right`, `left` or `colour:` will be reflected in the amphibian microworld.

The parallel use of textual and graphical user interfaces to elicit exactly the same behaviours and state changes has three purposes. Firstly, it introduces students gently to the language used to program the simulation, Smalltalk. Secondly, the parallelism with the GUI can be used to explain elements of the Smalltalk language in the context of a semantics that has already been explored and well understood via the graphical user interface. Finally, the fact that the simulation can be controlled equally well by either interface helps to establish the fact that the simulation exists independently of either user interface. This is a key point for the teaching later in the course of the details of a separable user interface architecture which enable students to implement their own GUI.

Having used a programming language in ways such as shown in the example above to elicit existing behaviours in the simulation, students are then shown how the programming language can be used to create new behaviours. To summarize briefly, this is achieved by showing students how to package up a series of message expressions such as those shown above in order to define a new method (in a class browser) that can then be added to the repertoire of any chosen kind of amphibian. The new behaviour will be immediately displayable in the existing simulation since the behaviour will be composed from existing displayable behaviours. Thus students can create new behaviours while still keeping in the bounds of the visible simulation.

Students are next shown how new behaviours, such as dances, can be associated with new variants of an existing class (e.g. a modified `Toad` or `Frog` class). Students also learn how to modify the effect of existing messages (e.g. `left`, `right`, `green`, `brown`) for instances of such a new class. Once again, all of the new student-created behaviours and instances of new classes will to be automatically displayable and visible in the simulation, without students explicitly having to attend to display mechanisms.

As the course progresses students are introduced to other microworlds, for example a simulation of an air traffic control system. Eventually they are able to construct new kinds of objects from the ground up and – provided they are subclassed from a displayable object and no new state added – the objects, their state and their behaviours will all automatically be displayed in the simulation, without students having to pay any attention to explicit graphical interface programming. This remarkable property follows from Smalltalk's separable user interface architecture, with which students are rapidly acquainted. This architecture is

particularly suitable for working with simulations, because the domain model and user interface can be developed or modified quite separately. More precisely, instances of objects that students create are visible in the simulation provided that any time their state changes, students arrange for the objects to issue a single message inviting any user interfaces that may exist to query their state and then mirror it graphically. The user interface must already know how to display the given kind of state, but provided students subclass from known classes and do not add state, this will happen automatically. If these conditions are violated, then the visibility of objects in the simulation will be lost, as students are encouraged to explore systematically. Ultimately neophytes are shown how to construct their own novel graphical interfaces, and their own domain objects and thus create their own simulations.

Hence the simulation approach is used by students not only to analyse, understand, and modify complex systems, but also to create their own simulations including their own separable and modifiable graphical interfaces. This both demonstrates the power of simulation and its embodiment in pure object systems.

8 GOALS FOR GUI IMPLEMENTATION

As outlined above, our pedagogy very clearly establishes the separation of domain model and user interface. Given this, for the notion of separable user interface architectures to mean anything to novices, they have to be able make concrete the abstract by eventually implementing their own user interfaces using a GUI builder, and to write all of the domain model and user interface code necessary to make these interfaces work. The most general goal of this part of our pedagogy was that students should understand, in detail, the advantages of allowing user interface and domain model development to be pursued independently. Specifically, our students should be able to:

- create their own user interfaces, and connect them to domain models, thus creating applications.
- understand and use the broadcast dependency mechanism and the messages involved;
- alter existing user interfaces built with a GUI builder;
- give a model multiple user interfaces simultaneously;
- reconnect a user interface to different domain models (i.e. to a different instance, or, to an instance of a different class where appropriate).

One of the earliest and most developed architectures in Smalltalk is Model View/Controller (MVC) [14]. MVC is an approach to application development that divides the application into the information of interest (model), the visual representation of that information (view), and the handling of user input (controller). This was the first widely used architecture for user interfaces that allowed models (i.e. application code) to be written quite independently of their user interfaces, and which allowed user interfaces to be modified, replaced or run several at the same time, without the need for any changes at all to the model. The model need not know whether it has a user interface, or how many user interfaces it has. These characteristics are the hallmark of a separable user interface.

In order to allow various kinds of flexibility, VisualWorks instantiation of MVC introduces a kind of impedance matcher or buffer class whose instances stand between the model and the various widgets. This class is called `ApplicationModel`. In effect, application models absorb the messiness of practical linking between model and user interface widgets, and translate messages from widgets in the `value/value:` protocol into the domain specific protocol of the model in question. The cost of this approach is that a lot of knowledge about both domain model and user interface needs to be built into the application model by programmer. Indeed, the application model may end up mirroring and hence duplicating much of the domain model. This architecture works very well for professional programmers. It allows tremendous

flexibility in the fine detail of how widgets can be used, and it keeps user interface logic out of domain code. However, even for experienced programmers, using the MVC mechanism [15] can be very daunting. For any given application there are will be multiple models, views and controllers. For example, the 'model' for even a small application will consist of the application model, and possibly multiple models from the domain model. The user interface will consist of many views (the graphical representation of widgets). Each widget will be a dependent of some different aspect of the application model rather than the whole user interface simply being a dependent of some domain object. Therefore, for the beginner, this complexity tends to obscure the essential simplicity of the separable user interface idea.

In a degree course aimed principally at first-time computing students, the complexity of the MVC architecture behind the VisualWorks GUI builder poses a problem that is not mitigated by the friendly and simplifying front end of LearningWorks, even with our principle progressive disclosure [10]. We did not want students merely to learn to use a GUI builder. We wanted them to understand and to work with separable user interfaces, to understand the architecture that makes them possible. To this end we devised a simplified version of MVC which we term MUI, Model–User Interface. MUI, is an architecture that allows beginners to easily create GUI applications by binding domain objects (models) to instances of user interface classes that they create using a simple GUI building tool. At each stage of the process extensive checks are carried out and the runtime environment ensures that problems do not result in cryptic Smalltalk exceptions but instead are reported in an understandable way (in the vocabulary of the course), after which a safe recovery is made.

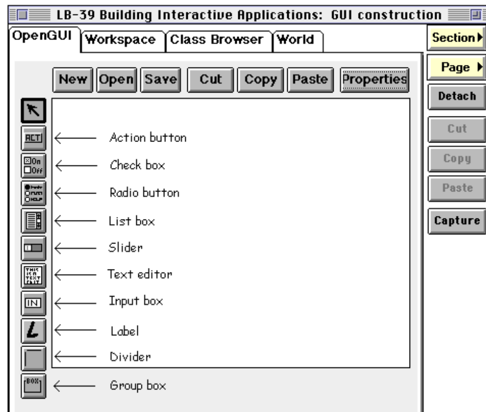
Before we introduce the MUI architecture to our students, they are taught about the broadcast dependency mechanism from which the *Observer* pattern is abstracted [16, 17]. Using this mechanism one object (the observer) is made a dependent of another object (the observed). They learn that to notify an observer object of any state changes the observed must include `self changed` message expressions in its state-changing methods. The model's only responsibility is to notify its dependents when its state changes. In that way the model need not take any account of what its dependents are, or indeed whether it has any. The responsibility is on any user interface components to respond appropriately to such notifications, and, where appropriate, to query the model about its current state so that they can update themselves suitably.

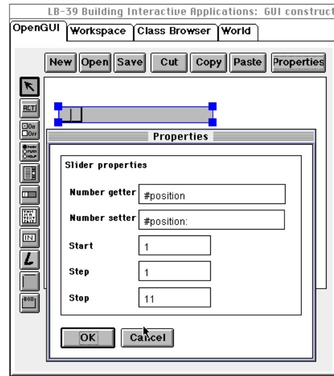
Our design goal for MUI was that an arbitrary model could be bound to an arbitrary user interface via this broadcast dependency mechanism. In other words, we wanted an architecture in which a user interface class could be made a direct dependent of a model rather than the individual widgets being dependent on value models in an application model as is the case with the VisualWorks instantiation of MVC. Furthermore, as long as the model provided the protocol expected by the widgets in the user interface it should simply work without any further 'glue' code being written by the user. In fact, the only user interface related code that appears in the model is the `self changed` expression mentioned above so that the user interface can be alerted via the broadcast dependency mechanism that the state of the model has changed. There are three basic components to the architecture which we subsequently outline: the OpenGUI tool, the user interface widgets and the test page.

This OpenGUI tool appears as a page in a LearningBook (see Figure 4). It looks like a fairly standard (if very simple) tool for laying out interface widgets and giving them property values. (This tool is subclassed from the drawing application used earlier in the course, so its user interface is familiar to students.) OpenGUI supports the following widgets: label, divider, group box, action button, check box, radio button, slider, text editor, input field and list box. We considered various ways of implementing menu buttons and other more complex widgets, but inevitably interface code ends up in the model and so they were rejected.

The amphibian microworld described earlier displays graphical representations of `Frog` objects. Their protocol includes the messages `position` and `position:`, which are used to get and set

the `position` instance variable representing a frog's position attribute. With this information students can construct an alternative user interface to control instances of the `Frog` class.





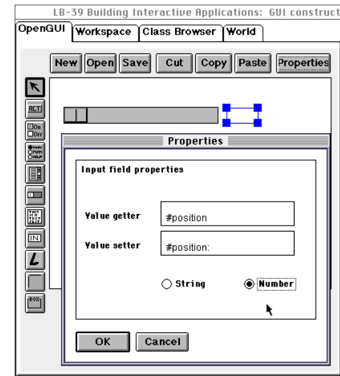


Figure 4 **Figure 5** **Figure 6**

In the OpenGUI page the student selects the slider drawing tool and draws the slider to the required size on the canvas. With the drawn slider still selected the user clicks on the **Properties** button to open a dialogue box which will allow the properties of the slider to be set. (See Figure 5.) As sliders can only work with numerical information the user is asked to supply the names of messages that will get and set a numerical instance variable in the prospective model (in this case a frog). Note that in the VisualWorks GUI builder the user would be asked to supply the name of a value model in the application model, rather than a message from the domain model's protocol. Other values requested are highest and lowest values that the slider can set in the model using the setter message, and the size of the increments the slider can make. To complement the slider an input box can be added to the canvas as in Figure 6.

Note again that the user supplies getter and setter messages from the model's protocol rather than the name of a value model in the application model. As we mentioned earlier some of our widgets are much simpler than the equivalent VisualWorks widgets. As you can see in Figure 5 the OpenGUI input field supports only two types – number and string. Also input fields are always editable. To make 'editable' another property would be trivial to implement but this was resisted in the cause of simplicity.

When saving the properties of a widget, OpenGUI ensures:

q that none of the fields in the properties dialogue box are blank;

q that any getter is a legal unary selector;

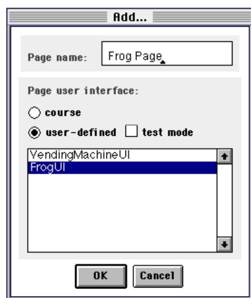
q that any setter is a legal single keyword selector.

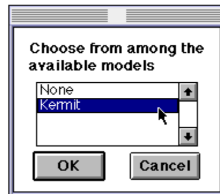
Before saving the user interface as a class, OpenGUI checks that:

1. the user interface is complete and consistent and the user is warned if they are using the same selector in more than one way in the same user interface;
2. the class name chosen, is either a new class name or the name of an existing student-authored user interface class.

The user interface class is then saved as a subclass of `OUGUIAbstractInterface`, itself a subclass of `ApplicationModel` in VisualWorks. Such a user interface class consists of a single class method, `windowSpec`, which specifies how the user interface is to be drawn. When an instance of the user interface class is opened, methods in the superclass `OUGUIAbstractInterface` (which hides the detail of implementation) dynamically create all the value models needed to support the MVC architecture that underpins MUI.

Attaching an instance of this new interface to a suitable model is achieved through the LearningBook's **Page** menu button. Selecting the **Add...** option opens up scrollable list of available user interface classes (Figure 7).





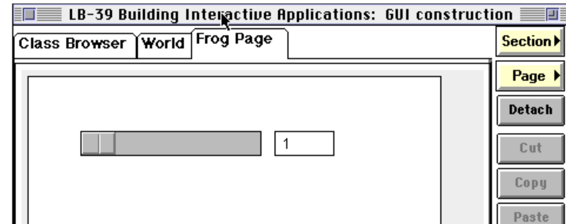


Figure 7 **Figure 8** **Figure 9**

After selecting the desired user interface class the neophyte programmer is then prompted to select an appropriate model from the section dictionary (Figure 8), in other words simply to select a section-local variable. This simple reference to a suitable model must have previously been established in the workspace in the same section.

To guarantee that the contract between user interface and model has been properly established, the MUI behaviour inherited by the particular user interface then carries out extensive checks on the model's protocol and degrades gracefully (see Figure 12) if the right methods are not present in the model, or if they return a message reply of the wrong type. If the checks establish that a contract between user interface and the model can be properly established, an instance of the user interface is created and inserted as the current page in the LearningBook (Figure 9).

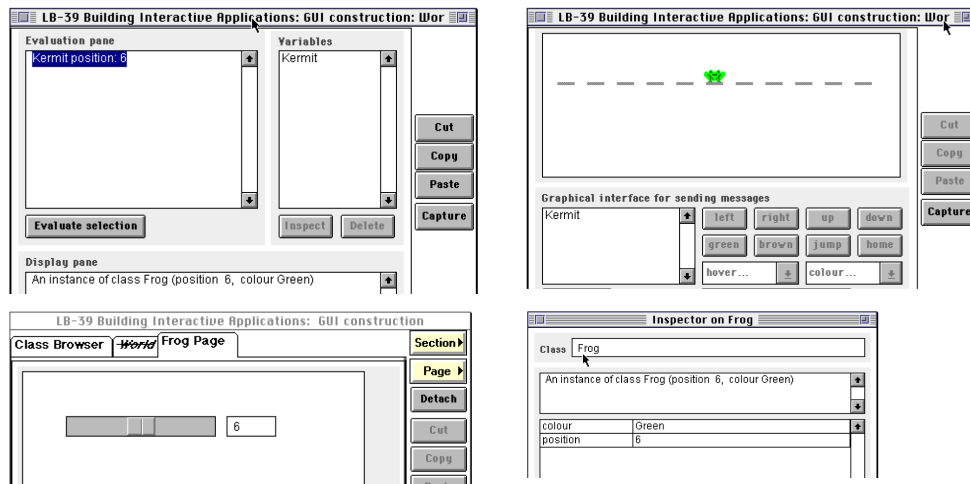


Figure 10

A model from the section dictionary can be associated with any number of user interfaces pages within the same section as the model. Therefore, by 'detaching' a Workspace and an amphibian World from the same section of the LearningBook, and placing them side by side with the new user interface page, students can view the effect of sending state-changing messages to the model from a Workspace, the state changes will of course be reflected in both user interfaces. Similarly, changing the state of the model from either user interface will be reflected in the other user interface and can be confirmed by inspecting the model from the Workspace. Thus reinforcing the notion of separable user interfaces. See Figure 10.

At any time the user can choose **Test mode** from the **Page** menu button to associate a new model (from the section dictionary) with the page's user interface (see Figure 11). This new

model need not be of the same class as the previous model, the only proviso is that it understands the required protocol and that the appropriate setter methods include the `self changed` message expression.

Exceptions and errors involving user interface classes and models in VisualWorks are next to impossible for the neophyte programmer to debug or reason about because of the huge number of complex classes involved. With the MUI architecture, students do not need to see MUI-specific classes or VisualWorks runtime GUI code to understand any errors associated with binding an instance of a user interface class with a model – problems are caught and reported in terms of objects and code created directly by the student. For example, when adding an instance of a user interface class as a page in the LearningBook, if the user interface discovers that the selected model cannot respond to the entire protocol needed by the interface, or if a getter method returns an object of the wrong class, details are reported and no attempt is made to open the interface, instead a test page is inserted into the LearningBook from where the user can select other pairs of local variables and interface classes. In Figure 12, the user has attempted to bind an instance of the `FrogUI` class to a model which is of class Account rather than of class `Frog` – `Account` objects do not understand the message `position:`.

Similarly, once a user interface is opened, MUI detects the following errors:

- that the get or set methods it needs to use are no longer understood by the model (i.e. the user has deleted the method since the interface was opened);
- the return value of a getter or the argument of a setter does not match the appropriate type (because the user has edited the method since the user interface was opened);
- that a method in the model causes an exception.

These errors are reported to the user and a test page is substituted for the current user interface page. Finally, if the method called by a widget results in an infinite loop and the user presses break, they are told which method was probably in the infinite loop and a safe recovery is made.

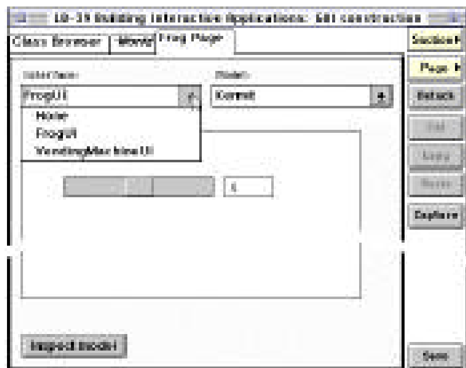


Figure 11

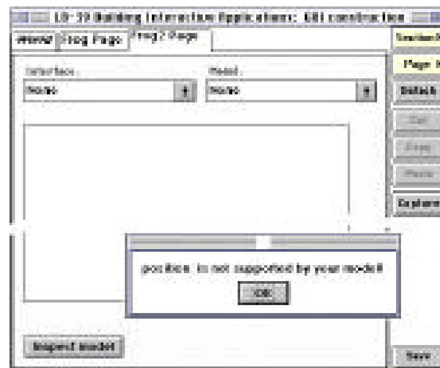


Figure 12

There are some significant restrictions in the current implemementation of OpenGUI. For instance, the requirement that a local variable be used to link a user interface to a model means that a collection of models cannot be linked to collection of user interface instances. And, of course OpenGUI is much simpler than a commercial GUI builder, in that it supports fewer widgets, and it assumes that a user interface can only deal with one model at a time (i.e. the views in the user interface only show information from one model). However, it is conceptually straightforward and simple for beginners to use. In effect, we have traded off some loss of flexibility with a tool that allows the unconfident to experiment concretely with all of the key concepts of separable interface architectures.

9 CONCLUSION

M206, *Computing: An Object-oriented Approach* is aimed at the needs of industry and departs from conventional introductions to software development by its object-oriented account of computing and its goal of producing graduates who can think in terms of complex, long-running software object-oriented systems. We have deployed a wide range of technologies to support learners grappling with fundamental concepts of object technology in a way that was appropriate to the distance mode and, in particular, to provide a transformation from novice to accomplished practitioner. We have achieved this because of our firm adherence to a consistent set of pedagogical principles and because of the basic soundness of the LearningWorks design and our principled use of it. As we have outlined, we have devised a clear pedagogy and constructed a programming and learning environment to match the pedagogy, using LearningBooks to package work, systems and tools. Most importantly we have devised and implemented the principle we call progressive disclosure. Hence, using LearningWorks and a wide variety of simulations and programming tools, novices can progress from using and extending systems through 'game play' microworlds, to programming in all its minutiae, through object-oriented analysis and design to graphical user interface design and implementation. And, while doing so they become proficient in a sophisticated programming environment, one recognisable by professionals.

References

[1] Woodman, M., Holland S. and Price, B., Pervasiveness of a Programming Paradigm: Questions Concerning an Object-oriented Approach, *Proceedings CS Education,* Dublin, 1994.

[2] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Reading, MA, 1983.

[3] Woodman, M. and Griffiths, R., Programming Language Choice for Distance Computing, in M. Woodman, *Programming Language Choice,* International Thomson Computer Press, London, 1996.

[4] 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, June 1996.

[5] Woodman, M., Law A., Holland S. and Griffiths, R, The Object Shop – Using CD-ROM Multimedia To Introduce Object Concepts. *Proceedings SIGCSE 97*, San Jose, February 1997.

[6] Poniatowska, B., Richards, M., Griffiths, R., Robinson, H. and Woodman, M., Organising Online Resources Between Web and Computer-based Conferencing. *Submitted to EdMedia 99*.

[7] Sumner, T. and Taylor, J., New Media, New Practices: Experiences in Open Learning Course Design. *Proceedings CHI '98*, pp432–439, Los Angeles, April 18–23 1998.

[8] Wirfs-Brock, R., Wilkerson, B. and Wiener, L. *Designing Object-oriented Software,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[9] Cook, S. and Daniels, J., *Designing Object Systems,* Prentice Hall Int., Hemel Hempstead, 1994.

[10] Woodman, M., Griffiths, R., Macgregor, M., Holland, S., and Robinson, H., Exploiting Smalltalk Modules In A Customizable Programming Environment, *Proceedings of ICSE 21, International Conference on Software Engineering,* Los Angeles, May 1999.

[11] Goldberg, A., Abell, S., and Leibs, D., The LearningWorks Delivery and Development Framework, *Communications of the ACM*, 40(10), 78–81, 1997.

[12] Wirfs-Brock, R.J. and Johnson, R.E., Surveying Current Research in Object-oriented Design. *Communications of the ACM*, Vol 33, No 9, pp104 – 124, September, 1990.

[13] Goldberg, A. and Ross, J., Is the Smalltalk-80 System for Children?, *Byte*, 6(8), August 1981.

[14] Krasner G. E., Pope S. T., *A Cookbook for using the Model View Controller User Interface Paradigm* in Smalltalk 80 Journal of Object-oriented Programming, Vol 1, #3 pages 26–49, 1988.

[15] Howard, T., *The Smalltalk Developer's Guide to VisualWorks*, SIGS Books, New York, 1995.

[16] Gamma E., Helm R., Johnson R., Vliffides J. *Design Patterns: Elements of re-usable Object-oriented Software*, Addison Wesley, 1994.

[17] Alpert S. R., *The Design Patterns Smalltalk Companion*, 1998, Addison Wesley, 1998.