Employing Object Technology to Expose Fundamental Object Concepts

Employing Object Technology to Expose Fundamental Object Concepts

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

Abstract

This paper explores technical issues in the design of programming tools, development environments, simulations, code examples, user interface frameworks and pedagogies for a university-level course on object-oriented software development. The course, M206 'Computing: An Object-oriented Approach' has been specifically developed for distance learning, and is enrolling over 5,000 students per year (average age 37) in the UK, Europe and Singapore. The course introduces computing via an object-oriented approach. M206 is substantial in extent, representing one sixth of a degree. It embodies a practical, industry-oriented view of computing and includes programming, analysis, design, and group working. Considerable effort has been invested in making the simplicity, consistency and power of object technology accessible to and capable of being applied by beginners. A diverse set of educational media, such as CD-ROMs, TV and the Web, have been deployed as learning resources. The paper describes the agenda for the course, its object-oriented pedagogy and our strategy for delivery. We explain measures taken to avoid misconceptions about objects, our analysis and design method, and the Smalltalk programming environment we have developed specifically for learners and which is crucial to our approach. The paper outlines how our adherence to the separation of view and domain model leads to technical innovations. Concluding remarks reflect on the benefits a reflexive strategy, both in education and training.

1 Introduction

The work discussed in this paper is strongly influenced by its context, which we will briefly outline. The Open University (OU) is the UK's largest university, educating nearly 10% of all UK graduates. Its courses are specifically designed and delivered via distance learning in the UK, Western Europe, and world-wide. The work discussed here arose from the development of M206 *Computing: An Object-oriented Approach* – an introduction to designing and writing complex software systems. This is a very large course – 440 hours of study during 33 weeks, worth one sixth of a degree and requiring fifty person-years to develop a new approach from scratch. It provides ordinary users of computer systems the resources to become developers of them and teaches object technology from the start. M206 is enrolling over 5,000 students per year; with average age 37 and typically in middle management with little or no programming experience. (Consequently, the social and educational impact of tens of thousands first learning about software in object-oriented terms will be immediate, and possibly dramatic.) From early 1999, some 600 students are taking M206 at the Singapore Institute of Management, and it will soon become available in Hong Kong and the USA.

When devising the syllabus for this course in the early 1990s, it was necessary to assess the technologies for software development most likely to be relevant at the end of the decade and beyond. To this end we reviewed and refined our plans in industry as well as academic fora. Industry made it clear that it needed people who could think in terms of complex, long-running software systems, not just in terms of simple input–process–output programs. From these considerations object and network technologies assumed central roles in the course and now strongly characterise its view of computing. We decided to adopt an 'objects-first' approach to computing [1] for reasons of power and simplicity. Following from this we chose Smalltalk-80 [2], an industrial-strength language offering the primary benefits that the language is based on just a few concepts [3] and its programming environments have the following properties:

- they are simple embodiments of object technology;
- they lend themselves to tailoring;
- they facilitate the production of student-alterable simulations.

The syllabus is industry-oriented. It 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 present our account of computing as isolated topics, but could offer an account encompassing most of the aspects involved in real systems. This is achieved through a syllabus covering the following topics in an appropriately integrated fashion: Systems Thinking, Network Computing (the Internet, Web, conferencing), Human–Computer Interaction (HCI), 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, the goal is that a student, among other things, will:

(i) have an 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 systems;
(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 and design.

As may be inferred from the above topics and learning outcomes, a course aim was to be very practical. A learning-by-doing pedagogy suitable for the distance mode was designed accordingly [4]. This required carefully focusing on two issues – how to best teach the concepts of object technology, and how to appropriately exploit diverse educational media to achieve that goal. Consequently, a fully integrated multimedia approach was adopted: 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 to be core.

In subsequent sections we describe the initial pedagogy for the course, which specifically addressed common misconceptions, 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 LearningWorks to support our pedagogy.

2 Principles of Pedagogy

The course begins with elementary computing and software vocabulary with a distinctly object-oriented account of systems made up of components that have state and message-activated behaviour. Indeed, we take great pains to establish a vocabulary and grammar for use in learning programming and subsequently learning analysis and design; we believe it essential that there be a single language of discourse for these apparently distinct activities. As part of learning to think like developers, students quickly progress to an exploration of two commonplace applications – a word processor and a drawing application. While we teach their use for practical purposes later in the course, the main pedagogic use is to make object concepts more concrete. Thus, for example, we explore the notions of object *state* and *behaviour* by discussing examples such as size and font attributes and how they are changed. Furthermore, we explain at a very high level the way in which an application can be modelled by a set of interacting objects sending messages to each other. In this way learners are encouraged to think about appropriate models of the behaviour they are used to. These ‘appropriate’ models should lead seamlessly to object-oriented programming.

Separation of concerns is an important principle of software development that is clearly important in well-designed object software. To be able to separate consideration of the user interface from the functionality of software we next introduce HCI. As part of the transition from user to software developer, students learn about the similarities and differences of different GUI operating systems and which are important and which are unimportant. They learn that names and icons (e.g. of buttons) 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 imprecise and gentle fashion, and having established HCI vocabulary, students next progress to an amphibian microworld of frogs and toads, which is discussed below. To avoid students thinking that programming is all about frogs and toads, we of course do introduce more mundane computer science examples.

Our pedagogy and the way we employed object technology to expose object concepts, especially with LearningWorks, was very much informed by an analysis [8] of likely misconceptions learners might form about object concepts and how these might be avoided. We were particularly aware that not only might students with previous programming experience confuse similar notions (e.g. variables as stores and variables as references) but the part-time tutors the OU depends on to provide students with teaching support might do the same. This is not because object concepts are intrinsically difficult, but because the subject does offer many opportunities for misconceptions to develop, which are hard to correct later and act as barriers through which later teaching may be inadvertently filtered and distorted. The problem of avoiding object concept misconceptions can be particularly acute in the case of distance education where it is often impractical to give frequent demonstrations or to provide immediate feedback to student queries. The misconceptions we concentrated on were ones we identified during the developmental testing of this course with prototype materials [7]. We highlight just a few from [8] here.

There is a temptation in early teaching to use examples in which only a single instance of each class is used. Some students become confused between classes and their instances. (Indeed, in some object-oriented languages there is no distinction or classes are compiletime entities.) To discourage this misconception, we ensure that examples use several instances of each class. We also adhere to a simple discipline of ensuring that all introductory examples use classes with more than one instance variable and that the instance variables reference objects of different class.

We have also avoided examples where an object behaves essentially like a database record, such as a music CD, storing information on the title, artist, tracks, etc. These types of objects overemphasise the *data* aspect of objects at the expense of the *behavioural* aspect and students may come to tacitly assume that all objects are inert records and fail to realise that the behaviour of some objects may alter substantially depending on their state. This early misconception can be avoided by using examples where the response to a message is substantially altered depending on the state of the object. For example, an `Account` object will refuse a debit request when an overdraft limit is reached. Such a request is 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 study can influence their thinking a lot. In many simple 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 messages which set state. Clearly, in a piece of general programming or a single teaching example, there is nothing in the least wrong with assignment. However, there is a danger that too much exposure to this way of changing state fosters the impression that real work in methods is done by assignment (and not by message sending). We have observed that very experienced students may pick up this impression and adopt a procedural style of coding. Thus, we use examples where the values of instance variables are not invariably immutable objects, but objects which themselves have state.

Another potential problem is to do with naming, reference and identity. Frequently in the traditional bank account example (which we ourselves use) two instance variables of different type, `name` and `balance`, are used in an intuitively clear way which is familiar to most people. However, in the minds of students with previous exposure to database concepts, a `name` instance variable can give rise to anxiety and misconceptions. They can confuse the `name` instance variable with the *identity* of the object, or with a variable that refers to the object, and so 'names' it (e.g. `myAccount`). These confusions can lead to further misconceptions, such as:

- 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 by mere discussion, the most effective strategy is to have students experiment with a set of disambiguating examples, summarised as follows:

Multiple assignments: get students to assign a single object to three variables at once and convince themselves that each variable references the same object by finding our that state changes effected via any reference can be inspected immediately via the other variables.

Re-assignment: get students to assign a different object (ideally of a different class) to one variable, and then explore by sending messages and inspecting the results that the variable refers to a different object, while the others 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.

Central to the pedagogy, and to the design of simulations and tools in the LearningWorks programming environment, is the notion of *progressive disclosure*. Students are limited in what they can see and do early on (e.g. initially very few classes are discussed or made visible in a browser) but we take care not to mislead students or avoid touching on complex ideas which are fully elaborated later. Indeed, once we have dealt with the basic concepts of Smalltalk programming (and have taught students how to use e-mail, conferencing and the Web) we rapidly progress to more elaborate ideas and appropriate visualisation tools for exploring further Smalltalk concepts: expression series, message answers, reference, variables, etc. Hence, to teach fundamental object concepts clearly we had to develop appropriate materials, programming environment and simulations to practice such activities.

3 Analysis and Design

A primary goal for our teaching of object-oriented analysis and design (OOAD) was that students should be able to use the same language they used when programming. Acknowledging that, in general, analysis is likely to uncover much richer structures than encountered when programming in Smalltalk, for an introductory course we considered it important to use a single coherent set of concepts and point out that design choices must be made about how to implement certain common relationships. For example, it is not automatic that *is-a-kind-of* will be represented in code via subclassing.

The OOAD approach we developed is loosely based on the CRC approach of Wirfs-Brock *et al*. [9] but with a flavour of the more formal treatment of associations given by Cook and Daniels [10]. This results in an overall 'feel' for students that is informal and intuitive 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, associations and instances. For instance, much of our early OOAD teaching is done via an example involving a hospital with wards, doctors, 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 informally as the rule that a nurse who supervises other nurses must be assigned to the same ward as the nurses she/he supervises. This rule is later 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, with the same instance of Ward that the given
instance of Nurse is associated with, via Is-Staffed-By.
```

Indeed, this insistence on expressing 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 our OOAD teaching:

> 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'.

This approach is pursued 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) and how instances may fulfil their responsibilities. Furthermore, the approach is underpinned with a number of key pedagogical 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.

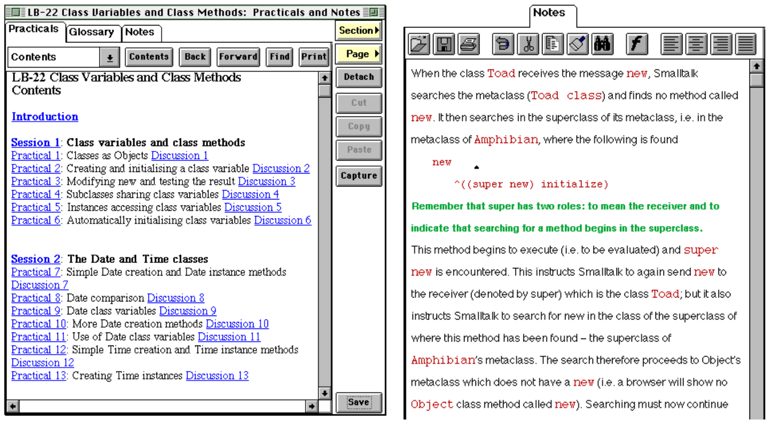
4 OU LearningWorks

The core course component used in teaching object concepts is OU LearningWorks, a Smalltalk environment which we developed [11] from work with Adele Goldberg and her colleagues on the LearningWorks programming framework [12]. After an analysis of how tester 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. Technically, these are binary representations of objects and classes. As far as learners are concerned, OU LearningWorks offers a set of LearningBooks whose user interfaces follow a book metaphor: each is organised into sections, and sections into pages. Each page is in fact a Smalltalk 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 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 always contains an HTML-browser page of practical activities and discussions, a glossary of terms, and a page for taking notes; see Figure 1. Subsequent sections of each LearningBook contain Smalltalk classes, programming tools – such as class browsers and workspaces – and microworlds (visual simulations) which students use to carry out the practical activities.

An overriding pedagogical requirement of the environment was that it should initially constrain novices and hide detail from them, as in Figure 2, but would progressively disclose the facilities and rich detail familiar to the experienced practitioner; this was especially important due to the complexity of object-oriented class libraries. The powerful LearningWorks *vision* mechanism [11, 12], amongst other things, allows a book author to define:

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

Hence as the course progresses, the LearningBooks provide increasingly more complex class browsers [11] which expose more of the complexity of the Smalltalk library.



LB-22 Class Variables and Class Methods
Contents

Introduction

Session 1: Class variables and class methods
Practical 1: Classes as Objects Discussion 1
Practical 2: Creating and initialising a class variable Discussion 2
Practical 3: Modifying new and testing the result Discussion 3
Practical 4: Subclasses sharing class variables Discussion 4
Practical 5: Instances accessing class variables Discussion 5
Practical 6: Automatically initialising class variables Discussion 6

Session 2: The Date and Time classes
Practical 7: Simple Date creation and Date instance methods Discussion 7
Practical 8: Date comparison Discussion 8
Practical 9: Date class variables Discussion 9
Practical 10: More Date creation methods Discussion 10
Practical 11: Use of Date class variables Discussion 11
Practical 12: Simple Time creation and Time instance methods Discussion 12
Practical 13: Creating Time instances Discussion 13

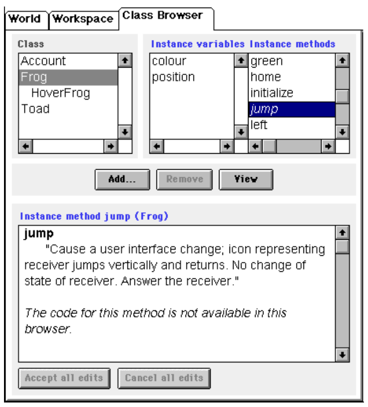
When the class `Toad` receives the message `new`, Smalltalk searches the metaclass (`Toad class`) and finds no method called `new`. It then searches in the superclass of its metaclass, i.e. in the metaclass of `Amphibian`, where the following is found

```
new
    ^((super new) initialize)
```

Remember that super has two roles: to mean the receiver and to indicate that searching for a method begins in the superclass.

This method begins to execute (i.e. to be evaluated) and `super new` is encountered. This instructs Smalltalk to again send `new` to the receiver (denoted by super) which is the class `Toad`; but it also instructs Smalltalk to search for new in the class of the superclass of where this method has been found – the superclass of `Amphibian`'s metaclass. The search therefore proceeds to Object's metaclass which does not have a `new` (i.e. a browser will show no `Object` class method called `new`). Searching must now continue

Figure 1. LearningBook First Section Pages



jump
"Cause a user interface change; icon representing receiver jumps vertically and returns. No change of state of receiver. Answer the receiver."

The code for this method is not available in this browser.

Figure 2. Simplest Browser

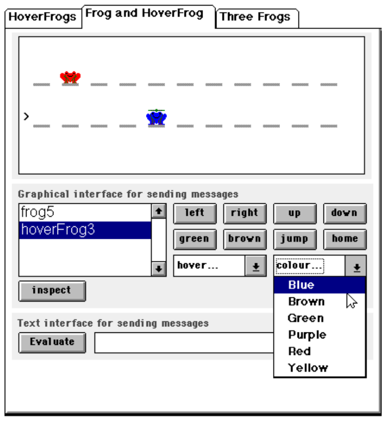
Next we describe one microworld which was designed to be a touchstone for students and tutors alike – a simulation with which they become intimate and using which they can reason about *all* object concepts taught in the course.

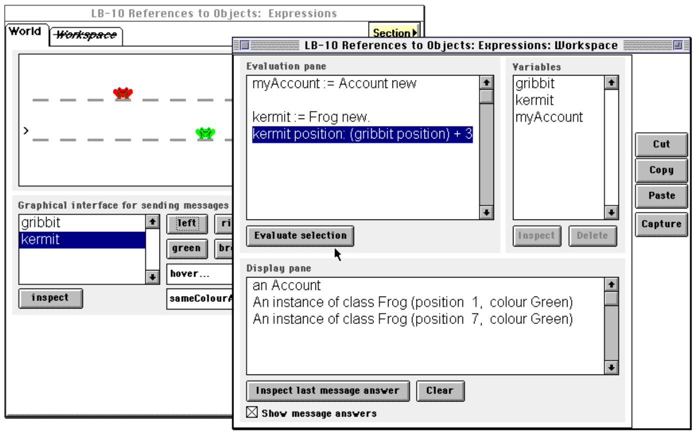
5 Simulations with Objects

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 shared reference model 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 cartoon-like world consisting of frogs and various other amphibians (two variations are shown in Figures 3 and 4). 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 action and menu buttons, students can look at the state of frogs, send messages to them, see how they behave in response, see how messages affect 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 essentially 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 right and left. Students select any of the objects in the microworld from a regular scrolling list (which they much later learn is another object!) and use the buttons to send the corresponding messages. This simple user interface not only allows straightforward message-sending to be visualised, it allows apparently advanced notions such as polymorphism to be demonstrated. For example, when a frog is selected and the **home** button is clicked (sending the message `home`) the receiving frog moves to the leftmost position, but if a toad has been selected, and so receives the message, 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 `hoverFrog3` to blue, it must receive a message `colour: Blue`; as can be seen in Figure 3, in the graphical user interface we have provided the button menu **colour**, whose menu items include the possible arguments of **Blue**, **Brown**, **Green**, etc.





```
myAccount := Account new

kermit := Frog new.
kermit position: (gribbit position) + 3
```

```
an Account
An instance of class Frog (position  1,  colour Green)
An instance of class Frog (position  7,  colour Green)
```

Figure 3. Amphibian World **Figure 4. World with Workspace**

To expose the notion of inheritance in which a subclass has more state and an extended protocol we have invented a new species of hovering frog, simulated by `HoverFrog` as a subclass of `Frog`. For example, `HoverFrogs` can hover vertically and have height in their state, and respond to messages that its superclass instances cannot, e.g. `Frogs`, 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 (bottom of Figure 3). 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.
```

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

6 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 the LearningWorks programming framework [12] we rely on 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 as they allow interactions between pages. 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 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, and any subclasses). 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 4 shows a LearningBook which is open at an 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 textually sending messages to amphibian objects via variables 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 user interface helps to establish the fact that the simulation's model 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 GUIs.

Having used Smalltalk to elicit existing behaviours in the simulation, students are then shown how the programming language can be used to create new behaviours. To summarise briefly, this is achieved by showing students how to package up a series of message expressions to define a new method (in a class browser) that can then be added to the repertoire of any kind of amphibian. The new behaviour will be immediately visible in the existing simulation since the behaviour will be composed from existing displayable behaviours. Students next learn how new behaviours can be associated with new variants of an existing class (e.g. a modified `Toad` or `Frog` class). They also learn how to modify the effect of existing messages (e.g. `left`, `right`, `green`, `brown`). Once again, as long as set messages are used for changing state (rather than assignment), 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 is added – the objects, their state and their behaviours will all automatically be visible in the simulation, without students having to pay any attention to explicit graphical interface programming.

The remarkable property that allows teachers to postpone the issue of GUI programming follows from Smalltalk's separable user interface architecture, with which students become 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 students learn how to construct their own graphical interfaces in a manner consistent with these simulations.

Hence the simulation approach is used by students not only to analyse, understand, and modify object-oriented systems, but also to prepare for a design architecture that allows them to build their own GUI applications. This demonstrates the power of appropriately designed simulations as learning resources for budding software developers.

7 Applications with Graphical User Interfaces

For learners, implementing their own GUI applications is a real fillip. As outlined, our pedagogy very clearly establishes the separation of domain model and user interface. To mean anything to novices, the notion of separable user interfaces must be made concrete by them eventually implementing their own user interfaces using a GUI builder, and writing all of the domain model. 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. In practical terms, they have to understand and use the broadcast dependency mechanism and become skilled at connecting instances of user interface classes to a variety of domain model objects.

The traditional Smalltalk approach is one of the earliest and most developed architectures – 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). 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. However, even for experienced programmers, using MVC [15] can be very daunting. In many professionally developed applications there may be multiple models, views and controllers. For example, 'the model' for even a small application may consist of an application model, together with multiple models from the domain. The user interface could consist of many views with each widget a dependent of some different aspect of the application model, rather than the whole user interface simply being a dependent of some domain object. For the beginner, this complication obscures the essential simplicity of the separable user interface idea.

In a degree course aimed principally at first-time computing students, the MVC architecture poses a problem for learners. 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 a framework that allows beginners to easily create GUI applications by binding 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 framework's 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.

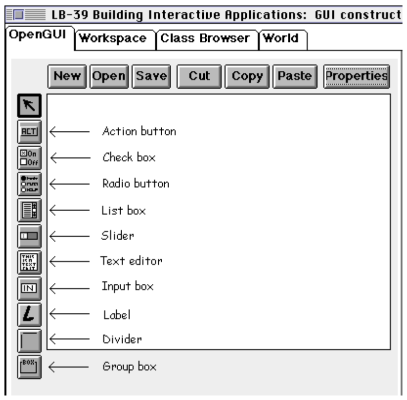
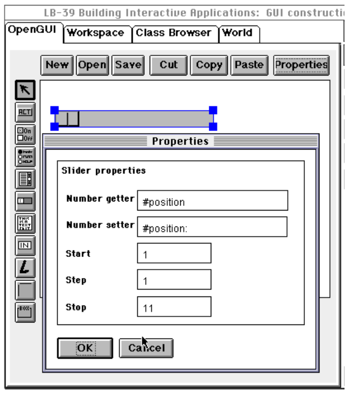


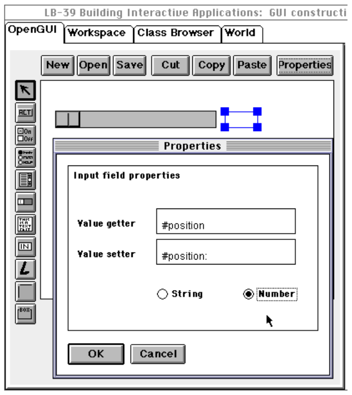
Figure 5. Simple GUI Builder



Slider properties

Number getter	#position
Number setter	#position:
Start	1
Step	1
Stop	11

Figure 6. Slider



Input field properties

Value getter	#position
Value setter	#position:

String [✓ Number]

Figure 7. Input Field

Before we introduce MUI 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 (hence the early de-emphasis of assignment). 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 the Smalltalk 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. Furthermore, as long as the model provides 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 code related to the user interface that appears in the model is the expression `self changed` in set methods. 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 (Figure 5). It looks like a fairly standard (if simple) tool for laying out widgets and setting their properties. (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.

To build an alternative GUI for controlling amphibians, learners should need to consider only the protocols of the domain classes. Their protocols includes the messages `position` and `position:`, which are used to get and set the `position` instance variable representing an amphibian's position attribute. With this information students can construct an alternative user interface to control instances of, say, the `Frog` class.

In the OpenGUI page the student selects the slider drawing tool and draws the slider to the required size on the canvas. With the slider still selected the user clicks on the **Properties** button to open a dialogue box for setting the properties of the slider (Figure 6.) 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. Other properties requested are highest and lowest values that the slider can set in the model, and the size of the increments the slider can make. To complement the slider an input field can be added to the canvas, as in Figure 7.

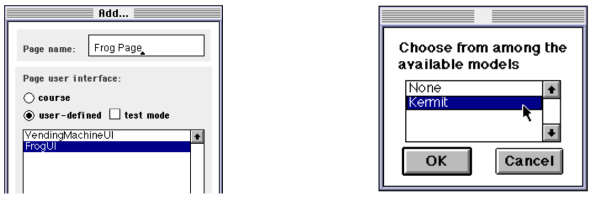


Figure 8. Selecting (a) GUI class and (b) Model

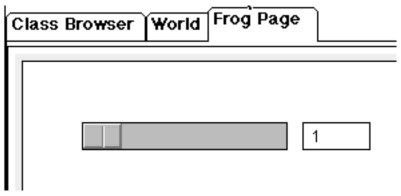
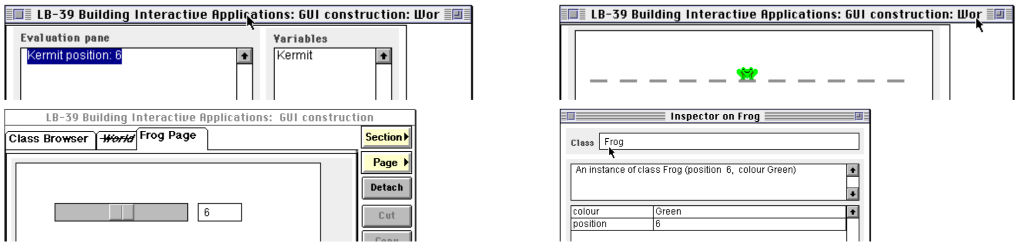


Figure 9. Amphibian Slider GUI

The user interface class is then saved – automatically as a subclass of the MUI framework. When an instance of the user interface class is opened, methods in its superclass dynamically create all that is needed to support the MVC architecture that underpins MUI.

Attaching an instance of this new user interface to a suitable model is achieved through the LearningBook's **Page** menu button. Its **Add...** option allows pages to be added to a LearningBook – programming tools, simulations, workspaces, or instances of student user interface classes, as in Figure 8(a). After selecting the desired user interface class the neophyte programmer is then prompted to select an appropriate model from the section dictionary as in Figure 8(b), i.e. simply to select a section-local variable. This simple reference to a suitable model must have previously been established in a 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 interface then carries out extensive checks on the model's protocol and degrades gracefully if the required 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).


LB-39 Building Interactive Applications: GUI construction: Wor
Evaluation pane
Variables
Kermit
LB-39 Building Interactive Applications: GUI construction: Wor
LB-39 Building Interactive Applications: GUI construction
Class Browser
~~World~~
Frog Page
Section
Page
Detach
Cut
6
Inspector on Frog
Class
Frog
colour
Green
position
6
Kermit position: 6

An instance of class Frog (position 6, colour Green)

Figure 10. Interaction of GUIs and Workspace in Same Section

A model from the section dictionary can be associated with any number of user interfaces pages within the same section as the model reference. Therefore, by detaching a workspace and an amphibian world from the same section of a 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 all user interfaces. Similarly, changing the state of the model from either graphical user interface will be reflected in the other and can be confirmed by inspecting the model from the workspace (Figure 10) – thus reinforcing the notion of separable user interfaces.

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. This new model need not be of the same class as the previous model, but is must understand the protocol used in the GUI and its set methods must include the `self changed` expression.

Usually exceptions and errors involving user interfaces and models are next to impossible for the novice 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 framework classes to understand any errors associated with binding an instance of a user interface class with a model; they only see classes limited to them by the vision mechanism.

8 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, 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 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.

At the time of writing quantitative analysis of feedback is still going on. However, by everyday criteria the course is successful: discounting drop-out, well over 90% of students passed the exam in the first year; the course has won a prestigious IT Award from the British Computer Society; potential employers have praised M206 as modern and relevant, and have in part praised it for *not* being like traditional computer science.

Though our setting is academic, it should be clear that object technology training with distance learning involves considerable up-front resource. It takes considerable time for a team to establish common principles and to design how they should be realised.

Our core use of LearningWorks may be a testament to object-oriented frameworks, but it is also an endorsement of Smalltalk as an embodiment of object concepts and as a language whose environments are both powerful, and to varying degrees, reflexive. Much of what we have achieved could be done for other languages, but a lot could not be. Object technology at its best offers power, flexibility, consistency and simplicity. These characteristics can and should be actively employed in the instruction of newcomers to object concepts.

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

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

[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. *Proceedings EdMedia 99*, Seattle, June 1999.

[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] Holland, S., Griffiths, R., and Woodman, M. (1997) Avoiding Object Misconceptions, *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, San Jose, February 1997.

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

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

[11] 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.

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

[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.