Exploiting Smalltalk Modules In A Customizable Programming Environment

Exploiting Smalltalk Modules In A Customizable Programming Environment

Mark Woodman, Rob Griffiths, Malcolm Macgregor, Simon Holland, Hugh Robinson

Computing Department
The Open University
Walton Hall
Milton Keynes, England MK7 6AA
+44 1908 274066
m.woodman, r.w.griffiths, m.d.macgregor, s.holland, h.m.robinsonopen@open.ac.uk

This paper appeared as:

Woodman, M., Griffiths, R. Macgregor, M., Holland, S. and Robinson, H. (1999) Exploiting Smalltalk Modules In A Customizable Programming Environment. Proceedings of the International Conference on Software Engineering, ICSE 1999 Exploiting Smalltalk Modules In A Customizable Programming Environment

ABSTRACT

This paper describes how we have extended a module structure of the Smalltalk LearningWorks to provide a programming environment deigned for very large scale technology transfer. The 'module' is what we have termed the LearningBook, a set of classes and persistent objects, including an HTML browser, programming and visualization tools, and microworlds. The context for this development is a distance learning university course in object technology which has enrolled over 5,100 mature students in its first year – making it the largest such course in the world. While promoting a systems building approach, we have successfully added support for programming in the small and the needs of the isolated novice. Two principles have applied: (i) the programming environment and its modules fit into a consistent framework for personal management of study and (ii) details of complex facilities, such as the class library, are progressively disclosed as knowledge and sophistication grow. The paper shows how these principles have guided the exploitation of LearningBook modules. To provide context, relevant academic background is given. Early informal feedback is reported and a project currently underway to observe in detail how thousands of learners use the Smalltalk programming environment is sketched.

Keywords

Object-oriented technology, Education, Technology Transfer, Smalltalk Programming Environment, HTML

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

ABSTRACT
This paper describes how we have extended a module structure of the Smalltalk LearningWorks to provide a programming environment deigned for very large scale technology transfer. The 'module' is what we have termed the LearningBook, a set of classes and persistent objects, including an HTML browser, programming and visualization tools, and microworlds. The context for this development is a distance learning university course in object technology which has enrolled over 5,100 mature students in its first year – making it the largest such course in the world. While promoting a systems building approach, we have successfully added support for programming in the small and the needs of the isolated novice. Two principles have applied: (i) the programming environment and its modules fit into a consistent framework for personal management of study and (ii) details of complex facilities, such as the class library, are progressively disclosed as knowledge and sophistication grow. The paper shows how these principles have guided the exploitation of LearningBook modules. To provide context, relevant academic background is given. Early informal feedback is reported and a project currently underway to observe in detail how thousands of learners use the Smalltalk programming environment is sketched.

Keywords
Object-oriented technology, Education, Technology Transfer, Smalltalk Programming Environment, HTML

1 INTRODUCTION

In this section we sketch some background that motivated the requirements for our programming environment and to set the scene for the technical developments described.

The work reported here is part of a large-scale research and development project to produce and deploy a new introductory course in computer science and software engineering at the Open University (OU). The OU is the UK's largest university and since it was established more than two million people have studied with it in the UK, Europe and world-wide. The OU's primary mission is to make higher education available to adults regardless of their personal circumstances and earlier educational achievements. Typically for a student to achieve an honours degree takes some six years of part-time study; the average age of students is 37. These factors were key influences on our development of the syllabus and pedagogy and on our design of our version of LearningWorks – as should become apparent. It is worth noting that while now widely welcomed, the planning decision at the stage in 1994 to embrace an 'objects first' approach and to choose Smalltalk as primary teaching vehicle was seen as a radical and controversial step [1].

An important influence in the design of OU LearningWorks was the nature of the closely coupled materials we deliver to students and hence the nature of the team responsible for producing these. The distance learning materials we have produced for students cover some 440 hours of study, constituting a sixth of a degree. The cross-media materials include some fifty illustrated text documents (around thirty pages each), associated software, web pages, nationally broadcast television programmes produced in collaboration with the BBC, the Smalltalk programming environment, communications software, and computer conferences. The integration of these materials involves the production and testing several thousand individual multi-media deliverable components and requires experts in all media fields.

Decisions about the use of one medium frequently affect the use of another. For example, analysis of feedback from testers of early versions of OU LearningWorks and draft study texts were used to refine the design of the way in which the various media would be integrated [2]. As a result of this analysis, world-wide web technology was put at the heart of a personal study manager for students. This in turn influenced the design of the programming environment, requiring the inclusion of an HTML browser in all LearningBook modules. Ultimately, this led to us moving much of our teaching of object-oriented programming from printed text to HTML in LearningBooks.

The course is called *Computing: An Object-oriented Approach*. This emphasizes both the sharp focus on object technology and its general applicability. This has an influence on the way we customized LearningWorks, for example, obliging us to design and implement tools that are representative of similar ones in other programming environments. Furthermore, the heterogeneity of our students means that the initial use of the programming environment must be constrained in a way that makes it very simple and restricted to use, and all tools must be uncomplicated, consistent and not allow the student to get into trouble [3]. The syllabus itself is under our control and we developed it taking into account what could be well communicated and taught using such means; the topics not only include object-oriented programming, analysis and design, but networks, operating systems, human–computer interaction (HCI), and group working. (Details of the syllabus and multimedia presentation can be found at www-cs.open.ac.uk/~m206/.)

Our goal is to move learners from being essentially *users* of software to being *developers* of software. Hence an overriding requirement of any programming environment we would use was that it should progressively and seamlessly disclose full facilities and detail that are familiar to accomplished practitioner.

Several particular aspects of the syllabus have a direct bearing on OU LearningWorks. We give prominence to the separability of domain model and user interface. One innovation is the teaching of an MVC-style of application development to complete beginners, providing practical experience of separable user interfaces right from the first practical lessons. This was achieved by developing a simpler version of MVC [4] called MUI (Model–User Interface) and the tools and abstractions to go with it. This is discussed later in this paper. Consequently, as also described later, we have designed a simple GUI builder which avoids the more powerful but complicated VisualWorks facilities [4] that underpin LearningWorks.

Another part of the syllabus that impacts the programming environment, particularly through the design of LearningBooks, is object-oriented analysis and design. Our approach has been loosely centered around the CRC approach of Wirfs-Brock *et al.* [5] but with a flavour of the more formal treatment of associations given by Cook and Daniels [6]. Within this framework, we underscore a number of characteristics that govern successful accomplishment in the practice of analysis and design that includes (i) the separation of concerns (user interface versus problem domain, already discussed), (ii) the acquisition and practice of dispositional skills in the identification of classes, associations, responsibilities and collaborations by exposing students to a range of problem scenarios (provided by a variety of systems, discussed later) and (iii) the importance of re-use within design. This latter characteristic has obliged us to provide a range of class browser tools and to have organized classes within LearningBooks in such a way that they bear scrutiny and modification to support these ideas.

The modular nature of LearningBooks has been crucial to how we have customized LearningWorks as a whole to meet the general requirements of the distance education context. As will be discussed more later, we have split the programming environment into a traditional Smalltalk image and source file and a set of LearningBooks. So far, we have distributed three versions of LearningWorks like this, each successive version providing additional behaviour either in the image or in the set of LearningBooks; in February 1998, at the beginning of the OU's academic year, the course went live to over 5,100 students mostly in the UK. (It goes live in Singapore in February 1999 and is being adapted by a variety of institutions, including in the USA, for presentation in late 1999.) There is almost no object-oriented programming experience among students and despite the large enrollment, healthy skepticism about object technology has been in evidence. Scrutiny of on-line conferences shows student and tutor approval of the LearningWorks system to be high. It has been both robust and easy to repair.

In the next section we outline the OU version of LearningWorks, essentially giving a flavour of the organization of our LearningBooks and some of their tools and systems. In Section 3 we look at the aspects of LearningBooks as modules and how we have exploited them to best suit our pedagogy. Subsequently we examine the various novel programming tools – the class browsers that support our progressive disclosure principle (and properly show class method inheritance), the new workspace, the new class reporter and class editor tools and, finally, the new GUI builder. The paper concludes with a reflection of the work and some comments on what changes are planned and how we intend to investigate just how neophytes gain competence in programming.

2 OVERVIEW OF OU LEARNINGWORKS

As mentioned earlier, primarily to keep the size of LearningBooks down to a maximum of a few hundred kilobytes, we have split the programming environment into a conventional Smalltalk image containing 'standard' classes, i.e. those VisualWorks classes permitted as a runtime systems for LearningWorks, and our own core course classes. By the latter we mean the classes for our own framework and all the programming tools described later, but not the domain classes for microworld systems. The first application is run automatically; it is a *launcher* which is metaphorically a bookshelf for the LearningBook modules. These are kept in their original course-team defined form, and in saved versions containing new classes or additional state representing the user's work. The launcher does not show a static set of LearningBooks, but provides a view on a particular directory structure where original, saved or user-defined LearningBooks are located. This is needed to allow updating of the environment by adding new LearningBooks. The user can choose to show different versions, as well as to set other preferences such as size of HTML text and colours for code.

Divergence from 'standard' LearningWorks

The OU LearningWorks environment was developed in parallel with the version developed by Adele Goldberg and her colleagues [7] with whom we collaborated; for simplicity we refer to it as standard LearningWorks (although in some areas we established a 'standard' first). A major aim of Goldberg for LearningWorks was to provide a framework to develop learning environments "in which to explore ideas about computing and software system architectures, making use of a programming language that supports dynamic object modelling and libraries of selected objects" [7]. Although we articulate the development of OU LearningWorks in terms of the pedagogic goals in connection with our users, much of what we have achieved is widely applicable to neophyte practitioners and ad hoc users who require an environment that supports the principle of *progressive disclosure*: that beginners be gradually exposed to concepts and tools and the detail of an environment that is itself a complex system. Our changes to LearningWorks arose from our particular context.

The first noticeable difference is the user interface. Metaphorically we place pages in a 'binder' which has an external back cover with buttons for frequent operations (like cut and paste) on its right hand edge, rather than have such buttons at the bottom of each page. We have also removed section 'covers' which support section *themes* – views of state shared among pages in a section which Goldberg *et al.* use extensively in their tools and microworlds. We have also introduced various elements of colour and control of font styles and sizes – both to signal context and to assist visually impaired users – and we limited the size of windows and pages. Moreover, we have tended to use textually labelled buttons rather than icons. Examples are shown in Figure 1. These changes have been made for four main reasons:

- ❑ to simplify user interaction and to thereby fit the needs of distance education, and our syllabus;
- ❑ to realise our pedagogical structure;
- ❑ to simplify navigation within and between LearningBooks;
- ❑ to economise on prolific use of screen real-estate.

Next, we comment on the main LearningBook types and to what extent we used them.

Types of LearningBooks

The main LearningBook type is the *project book*, whose user interface implements a notebook metaphor in which books are organized into sections and sections into pages. Figure 1 shows one example of a project book. In terms of Smalltalk, the books are modules containing variables, objects and classes which can plug into an executing image. Class names have (as usual in Smalltalk) global scope and are loaded into the Smalltalk image when a LearningBook is open; such classes are deleted when the LearningBook is closed. In essence, pages show the user interfaces of applications. Furthermore, both sections and pages may have arbitrary state associated with them, particularly a dictionary of *local variables* which are mostly used in workspace pages and in microworld systems. Except in a few introductory and limited microworlds we invariably use section-local variables and so subsequent discussion of local variables refers to these.

In some instances, we had to commit to writing about certain tools before they were finalized in standard LearningWorks. This was a primary reason for not adopting the standard *inspector book*. Consequently, we developed our own inspectors. (Indeed, because of the collaboration with Goldberg, the look of the inspector pages in the standard inspector book are now similar to the our inspector windows.) The prescribed form of inspector in OU LearningWorks is to give the class, print-string and attributes of the class's state – instance variables and, if a collection, elements of the collection. An example of our inspectors are given in Figure 2 which shows the state of a simple instance of the class `Frog` (the print-string and the values of the two instance variables. For pedagogic reasons, we do not allow the state of objects to be changed in an inspector.

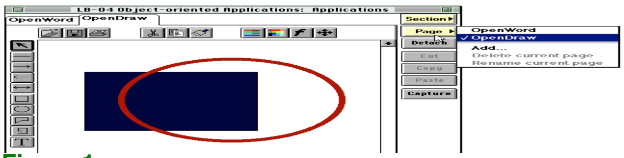
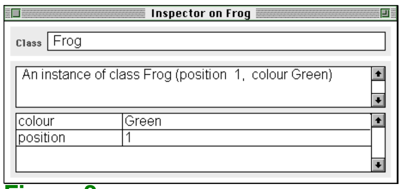


Figure 1

We do use debugger books and consequently this is our only use of section themes – the context stack at the top of each page of this single-section book. We are concerned with one course so do not use *course books*.


Inspector on Frog

Class: Frog

An instance of class Frog (position 1, colour Green)

colour	Green
position	1

Figure 2

LearningBook structure

As mentioned earlier, after an empirical study we let the needs of the neophyte practitioner dominate the design of LearningBooks via their user interface rather than their software structure. We concluded that the notebook metaphor would be a primary lever for the learner, and therefore, we adopted a consistent organization in which the first section of a book should contain practical exercises and discussions of them, a glossary of relevant terms (both as HTML browser applications) and a simple word processor for taking notes. Subsequent sections are organized to match the teaching strategy for the particular course chapter. So for example, the LearningBook for Chapter 22 (**LB-22**) which covers class variables, class-instance variables, methods, and the classes `Date`, `Time` and `Character`, has four sections. The first is **Practicals and Notes** which is described next; then there are the sections **Class variables and methods**, **Date and time**, and **Character representation**, with the obvious relationship to the topics covered. The pages of the second and subsequent sections contain the programming tools and microworlds with which students interact to pursue their studies – guided by the practical exercises and discussions of the first section. The general structure of the practical work in LearningBooks is an Introduction that links the work to the paper-based material, and a number of Sessions which break up the work; within each session are pairs of Practical exercises and Discussions (this is software engineering, so there are no 'solutions'!). Figure 3 gives a sample pages from a first section.

Generally student users are encouraged to detach the Practicals page – as if detaching a page from a notebook. Normally, detaching a page does just that: it places the page in a separate named window on the desktop, leaving only the page tab scored through in the notebook. However, for the HTML pages only, the page is cloned as a window on the desktop. We found this was necessary so that a student could consult the text of both an exercise and its discussion (and even copy from either) while interacting with the tool or microworld in another section. Even if not using an HTML browser for instructional purposes, this detach-a-copy facility can be generally useful. (It is possible that a future version of the environment will support this facility as generally possible behaviour for pages.)

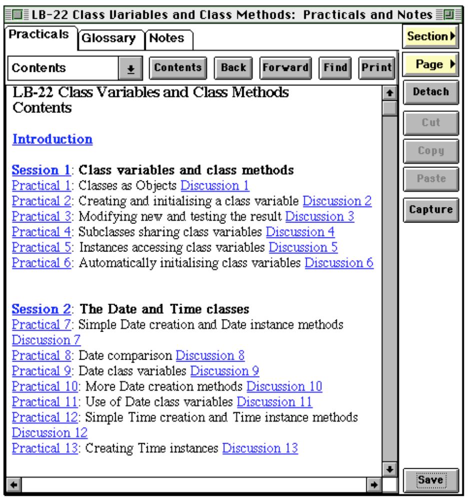
The constancy of our LearningBook organization is reassuring to beginners, but it has a number of advantages besides:

- ❑ It decouples sessions of practical programming ('lab sessions' in a conventional setting) from sections.
- ❑ It frees an author to group together microworlds, tools, etc. to fit a teaching strategy.
- ❑ It facilitates detaching, moving and the navigation of pages and windows, by keeping section and page changes to a minimum.

Microworlds

LearningWorks encourages the use of microworlds to motivate learners [8] who use and modify existing small systems rather than having to program from scratch. To provide a very simple, memorable, shared source of examples for virtually all object-oriented concepts, we developed an amphibian microworld in which instances of the classes `Frog` and `Toad` (or their subclasses) could be represented graphically. Later, once inheritance has been taught we have students redesign these classes to be concrete subclasses of an abstract class `Amphibian`.

Our requirement to provide students with a pedagogically simple and consistent learning environment turned out to demand some unexpected sophistication and new features in the design of the OU LearningWorks programming environment. This sophistication arose from pursuing a simple set of pedagogical requirements consistently to their conclusion. For newcomers to computing, objects like numbers and strings are somewhat atypical and abstract entities, and may not be pedagogically the best examples with which to introduce basic object concepts. We opted instead for initial graphical microworlds populated by cartoon-like depictions of concrete entities (such as frogs, toads, etc.), whose class and state is visibly obvious, whose every state change is visible, and on whom the effect of all messages is plain to see. We further required that any part of the microworld could be controlled equally well either by a GUI interface using selection, button presses and menu selection – that is, by a user interface that novice programmers but experienced users would find relatively straightforward – or equivalently by evaluating Smalltalk expressions.



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

Figure 3

Figure 4 shows just one of many microworld states involving frogs, toads, and the imaginary subspecies of frog, the hoverfrog. Hoverfrogs have a height attribute and can move up and down, thus hovering in the air! As can be imagined, simple button commands correspond to unary messages in the protocols, while any commands involving a menu selection correspond to messages requiring one or more arguments. A single-line input field is provided for simple Smalltalk expressions, and in some circumstances an output field is provided to show message answers. As already noted, the amphibian microworld (in fact a variant with all the buttons but no input field, see Figure 5) provides a simple, memorable, shared source of examples for all object concepts encountered in the course.

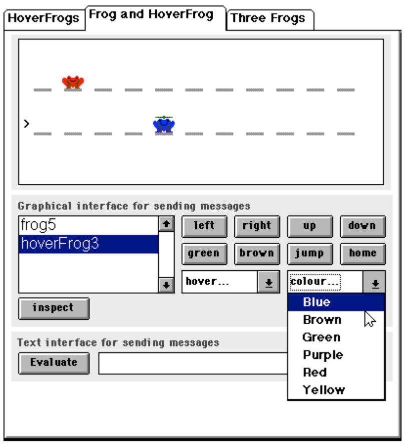


Figure 4

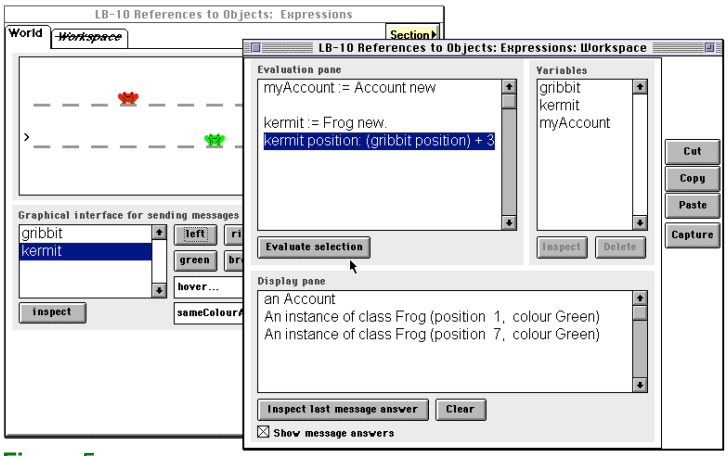
The duality of control just described (GUI buttons and Smalltalk text) allows HCI concepts to be used to provide useful concrete metaphors and explanations for otherwise abstract aspects of syntax and object behaviour. Unfortunately, the practical impossibility of an input field for programming and the limitations of screen real-estate mean that a separate workspace is needed – as a different application in a separate page. (LearningBook section state was indispensable in addressing this issue, as explained next.)

As described so far, the microworld approach does not differ very much from, for example, Goldberg and Ross's box world, and other introductory simulations. However, our wish to extend this approach to deal with assignment and object creation and destruction in a manner consistent with the way in which objects, messages and state are depicted provided important motivation for the new workspace features described next. This requirement had a significant impact on how LearningBooks are implemented to allow interaction between microworlds and any workspace in the same section. A second aspect of microworlds that influenced how new environment features were provided was the need to allow beginners to gain practical experience of a separable user interface architecture as early as possible. Hence the microworld design had to take account of our MUI (Model–User Interface) architecture and GUI builder (see below).

Workspaces

After much prototyping, the pedagogical requirements and the behaviour of the local variables of LearningWorks led us to a novel design for a workspace. We rejected the simple, traditional Smalltalk text pane (in which 'print it'/'show it'/'inspect it' commands are available) in favour of a more elaborate user interface that provided separate panes for (a) an Evaluation pane for typing in, selecting and evaluating expressions, (b) a Display pane to show the textual representations of message answers (i.e. print-string texts), and (c) a list of inspectable local variables. Figure 5 shows the ubiquitous amphibian world (without input field) in the background and our workspace tool in the foreground. Note that the local variable `myAccount` has been created to refer to an instance of the class `Account` but naturally does not appear in the amphibian world to which the class has no relevance. The variables, `kermit` and `gribbit`, however, refer to instances of relevant classes and so are shown and can be manipulated as the sample code demonstrates. It allows the novice to explore many of these somewhat abstract concepts quite concretely and with the truth of situations automatically reflected.

To facilitate the visualization of assignment, object creation and destruction, our amphibian microworld was built not as an arbitrary application, but, in effect, as a specialized graphical view of the dictionary that held all variable assignments in the local scope of the section in question (a fact that is of key pedagogical importance in later teaching of the concepts of assignment, reference, variables, dictionaries, etc.). Hence the structure of our LearningBook modules has a key bearing on the design and interaction of tools in the environment. Objects of any class can be created on any workspace page using the evaluation pane, even though a microworld in the same section specializes its graphical view of the local dictionary to display only objects of certain classes of interest (e.g. frogs and toads). Simply creating an object of a relevant class in a workspace and assigning it to a local variable causes its graphical representation to appear automatically in any interested microworld's graphical view. All entries in the local variable dictionary (referring to objects of any class) are also explicitly shown in a dedicated pane of the workspace. Any reassignments of any local variable to any object of any class are automatically updated in the view. In particular, if the student reassigns variables so that a particular displayed object has no remaining references to it, the automatic garbage collection of that object will be graphically dramatized in its immediate disappearance from the microworld. Imagine the visual effect of losing a reference to an aircraft in an air traffic control simulation! Hence we provide a microworld for such a simulation.


```
myAccount := Account new

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

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

Figure 5

This architecture of including workspace and a world together in a section, and so having shared access to section-local variables, is important to the construction of microworlds and to the context-sensitive characteristics of the programming tools. Not only does it allow beginners to make rapid progress early, it provides a straightforward modular structure for the designers of applications (microworlds in our context) and tools. So, for example, we have been able to provide a microworld that is a simulation of air traffic control in which planes disappear from view under certain circumstances; the problem is caused by a poor protocol for the class used in implementing the airspace and its replacement is trivially handled in the microworld without changing its design.

In the next section we further consider the use of LearningBooks as modules for our programming environment.

3 LEARNINGBOOKS AS MODULES

Because the facilities and limitations of LearningBooks as modules significantly affect how the environment and its tools are designed, we now consider several important aspects of LearningBooks.

Vision

The principle of progressive disclosure which has guided much of our exploitation of LearningBooks is of particular importance in a sophisticated programming environment containing a range of powerful tools and an extensive class library. Given the complexity of commercial programming environments, the Learning-Works environment proved ideal for such a philosophy by allowing us to dynamically customize it via the classes that successive LearningBooks load and by controlling access to those classes.

As already mentioned, LearningBooks can and usually do contain class definitions that are loaded into the Smalltalk image when the LearningBook is opened and deleted from the image when the LearningBook is closed. This class loading mechanism is extremely useful as it allows different LearningBooks to load different versions of the same class into the image. Indeed, we can even load in different (progressively more complex) versions of inheritance hierarchies when we deem it pedagogically necessary. In addition to this, LearningWorks provides a class and method scoping mechanism for LearningBooks, called the *vision*. This powerful mechanism amongst other things allows the LearningBook author to: import classes from other LearningBooks; specify which classes in the image are visible to the debugger and the class browser; specify for each visible class those methods whose code can be viewed and edited; specify for each visible class those methods whose code cannot be viewed. The vision for a book is set up when the book is created using a simple declarative language that names books, sections, etc. and binds them to instances of classes and user interfaces. For example, early in the course the vision of books is limited; i.e. restrictions on users are more severe than later on. Towards the end of the course very few restrictions still apply. As a specific example, take the ubiquitous `printOn:` message that generates textual representations of the state of classes of objects. It needs to be available throughout the course but its code should not be seen until after studying streams (about half way through in our pedagogy).

We found this scoping mechanism extremely useful, especially the ability to import scope from another LearningBook as we wanted the class browsers in each LearningBook to progressively disclose more of the class library as the student worked through the course. Early versions of standard LearningWorks did not fulfill all our requirements: we needed a section-based scoping mechanism because as a student works through a LearningBook, section by section, we wanted the class browser in each section to progressively disclose more classes or more methods in the classes. We overcame this limitation by providing our class browsers with filters that refine the book-based scope. While not ideal, this addition has satisfied our immediate needs.

Separation of image and LearningBooks

As already mentioned, the environment is delivered as an image and a set of LearningBooks, with the choice between what classes are in the image and what are in LearningBooks being determined by minimizing the size of the latter. (Keeping LearningBooks small also means that there load time is minimized, an important usability factor.) The image is a stripped down version of a VisualWorks image. The arrangement works satisfactorily as Smalltalk classes and objects can persist outside of the image in binary files – LearningBooks are examples of such files. When such a binary file is loaded into the image a record of the classes loaded is written to the image's *changes file*. This record is also used as the source for the text of methods displayed in class browsers. In the OU LearningWorks system the changes file is created when a user loads a LearningBook and deleted when the book is closed. This reflects the fact that the core image is never permanently changed – if a user creates a class in a LearningBook and closes and saves that LearningBook the image returns to the same state that pertained before the student opened a book. The change, a class creation, for example, is recorded in the actual LearningBook itself when that book is closed and saved.

One of the weaknesses and strengths of traditional Smalltalk environments, is that the programmer-user can change the image in anyway she or he wants. This approach leads to great flexibility, and makes it very easy to customize basic system functions quickly (e.g. you can change the window system and even the compiler to act in ways that you want). However, if a novice makes mistakes, and saves the image, the image can easily become corrupt in a way which is difficult and time-consuming to repair. For beginning programmers we thought it sensible to sacrifice flexibility for greater safety. Hence, the LearningWorks image cannot be saved, and therefore can never be corrupted. Similarly, we have arranged that the original LearningBooks supplied with the environment are always available for the user to return to if they need to abandon the book they were working on. So, the worst that can happen is that the student might create some classes in a LearningBook that corrupts the environment while that book is loaded. The current absence of modular change-logging for LearningBooks may be problematic for the student, who might not know the cause of an error. Provision of a log like the image's change file for each LearningBook to allow the student (or their tutor) to replay what happened and find the cause of an error would be ideal. This functionality is provided by a suite of classes being developed as part of a project to record and study how learners use the environment, which is sketched later.

In summary, the LearningBook and image structures we have used provide a highly modular framework which protect students from misusing the system while providing a good working environment. The image and the provided LearningBooks are sacrosanct modules, they cannot be changed or altered by the student through everyday use of the Smalltalk environment. The only modules the students can change in a persistent manner are the LearningBooks that they create and save themselves. Also, as we have used them, each

LearningBook is modular in respect to other LearningBooks, that is any change made to the state of one LearningBook cannot change the state of another LearningBook. Also, users know that any change to the state of the image is (a) not persistent (b) only due to the present LearningBook.

Loading and deleting classes

The loading and deleting of classed as LearningBooks are opened and closed has, unfortunately, resulted in major restriction on users – that only one LearningBook be open at once. When a LearningBook is closed the classes and objects that were loaded with the LearningBook are deleted, as are any classes that were created and any objects that were created. That is, the image is returned as close to the state it had before the book was loaded as possible as objects in the environment may have changed the state of the image while a book was one. (We can only be certain of getting back to pristine image state by quitting and restarting.) The general problem is having more than one version of a class. For instance, book LB-X might use the class `Amphibian` whereas a subsequently opened book, LB-Y, might require changes be made to `Amphibian`. While a pedagogy could cope with different behaviour of LB-X due to the changes effected in LB-Y, matters would be less than straightforward if either book, deleted `Amphibian` as it closed before the other.

We did not want to consider changes to class naming in Smalltalk, so are resigned to this constraint. However, we have experimentally implemented 'safe' books that have no impact on the class structure, we have deferred work on this at present. After a certain point in the course, i.e. in certain LearningBook modules, students may add their own sections and pages (see Figure 1). Programming tools of various type can therefore be introduced into a LearningBook that did not originally provide them and so to provide safety, it is likely that we would have to supply mechanisms to propagate changes to classes across a range of inter-dependent LearningBooks.

The constraint causes a significant problem when updating the environment because so much of the core environment is in the image. As a result of including most of the classes needed by LearningBooks for the HTML browser, the microworlds and the programming tools, problems found with the deployed system are difficult to deal with because of the current limitations of the Internet infrastructure which would make network distribution of full images error-prone and expensive. We have provided no means to save a changed image that is at the heart of the programming environment and its practical immutability is a significant problem when there is a requirement, as we have, to update over five thousand users when a bug-fix is implemented. Currently we have to provide completely new versions via CD-ROM; the image is between 4–5Mb and even with compression is too much to for students to download from their Web site. Fortunately, it is possible to make temporary or seemingly permanent changes to ease this inherent difficulty. A text file containing initialization code is read when LearningWorks starts. This arranges for the launcher to open and sets fonts and their sizes for the environment. If a bug-fix can be provided by installing a replacement class, then the initializing text file can be modified to read in what is essentially a patch. Similarly, the LearningBooks themselves can include classes that temporarily replace those in the image.

In the next section we describe some of the other programming tools we have implemented.

4 NEW PROGRAMMING TOOLS

In this section we briefly review the programming tools we have introduced in the OU LearningWorks environment. The dominant principle of progressive disclosure can be characterized by the slogan *eventual empowerment*; i.e. by the last chapter of a course supported by LearningWorks, by the last LearningBook, a student or trainee should be able to access all parts of the Smalltalk environment. The modular nature of LearningBooks has been essential in achieving our goals in this respect.

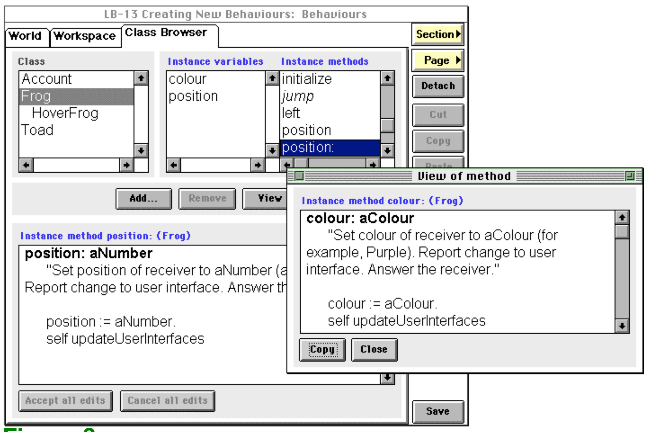
Class browsers and viewers

Common Smalltalk class browsers make it difficult to provide systematic teaching material. In VisualWorks, for example, the main 'System browser' simply exposes the student all of the classes in the system at once – many hundreds of them. To find the particular classes needed in a practical exercise needs a good knowledge of the 'search and find' techniques used by experienced programmers – and this before even knowing how to write the simplest expressions. Also, once a class is found, the novice is presented with a plethora of details – some of which are not covered until many months into the course (if at all). To get round these problems the OU LearningWorks browsers appear in LearningBooks in increasing degrees of sophistication and complexity as they are needed. And just the classes needed for that set of practicals, or which have been, are visible in the browser – due to the ability to define the vision of a LearningBook such that a browser can only 'see' certain classes, and within those classes can only see or allow access to certain parts of the class. Hence Figure 6 shows a browser from early in the course when only the classes `Account`, `Frog`, `HoverFrog` and `Toad` have been formally treated. None of the standard Smalltalk classes are visible (not even `Object`) although all can be used.

The browsers use text styles to indicate what a user can do with some element of a class. For example, if the name of a method is in plain style, the browser will show its code but not allow change; if the name is in italics only the initial method comments will be show; only if the names is in can the method be change.

Furthermore, our browsers show just enough detail of a class to allow students to carry out particular exercises. For example, Figure 6 shows only instance variables and instance methods, because at that stage of the these are all the students know about. Later browsers allow students to use radio buttons to switch between the instance and class side of the chosen class in the browser. Also shown in Figure 6 is a *view* window. We did not want users to have to start up different browsers to read different parts of a class definition simultaneously, nor did we want novices to become confused by the possibility of seeing one version of, for example, a method having changed and recompiled it. Therefore we have provided the **View** facility which essentially inspects a selected item; for example a user can view a variable and its comment, or a method and its code, or all of a class.

The later browsers begin to look more and more like traditional browsers. A conscious design goal was to provide evolving tools that would provide students with a good basis for using a commercial toolset later in their career. For this reason we did not employ the standard LearningWorks browsers, which are of quite a distinctive style and use a significantly different user interface approach using themes [7]. We have followed the lead of standard LearningWorks by not providing the traditional controls for programming tools. So we have provided buttons for accepting (compiling) code, for copying and pasting, for finding methods and for adding or removing items from a class definition. Only later versions of more sophisticated of our browsers provide the **Find** button (see Figure 7), **Edit** button (see Section 4.2) or button for filing-in or filing-out classes (not discussed). The **Find** menu button allows a user to find references to the selected class, references to a particular variable (the one currently selected in one of the scrolling panes), references to methods and references to the classes that implement particular methods.



Instance method position: (Frog)

```
position: aNumber
    "Set position of receiver to aNumber (a
Report change to user interface. Answer th

    position := aNumber.
    self updateUserInterfaces
```

Instance method colour: (Frog)

```
colour: aColour
    "Set colour of receiver to aColour (for
example, Purple). Report change to user
interface. Answer the receiver."

    colour := aColour.
    self updateUserInterfaces
```

Figure 6

The **Add...** button produces a version of a dialog box whose full power is only progressively disclosed, in line with the facilities of its browser. The fullest version allows any of the following to be added: a subclass, an instance variable, an instance method, a class variable, a class method, and a class-instance variable. A class addition and all the variable additions result in the user being prompted to document the item with a comment; variable commenting is often not provided by standard tools in programming environments.

Another important facility provided by the **View** button is to show the metaclass hierarchy. It's an unfortunate aspect of traditional Smalltalk class browsers that when context is switched from instance variables and methods to class variables and methods that the hierarchy remains the same. The context switch is actually from the class definition to the metaclass definition, which the hierarchical view should reflect. This user interface failure contributes significantly to the misunderstanding by both novices and experienced programmers of class method inheritance. The problem is characterized as the *where is new defined?* syndrome. Mostly because of limited screen real estate when deep in a hierarchy, we reluctantly concluded that we must adhere to this aberrant user interface design but we have provided a way of viewing the relevant metaclass hierarchy with the view facility: selecting a class method and clicking on the **View** button produces a view of the metaclass with any classes that define the method in bold. Figure 8 shows the inheritance of `new` from the metaclass of `Amphibian` (i.e. `Amphibian class`) via the metaclass of `Object` (`Object class`), `Class`, `Class Description` and finally `Behavior`. Hence it is absolutely clear that the class and metaclass hierarchies intersect and the *class* method `new` is found as an *instance* method of `Behavior`.

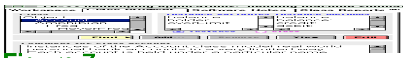
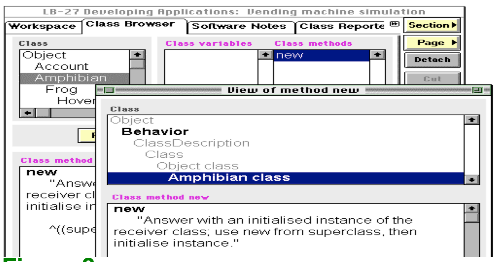


Figure 7

Class editor

Our analysis of how novices comprehend classes in Smalltalk led us to a requirement that the environment allow a class be viewed as a whole. This can be done by selecting a class and clicking the **View** button. To allow changes to be made while considering a whole class, we have provided a *class editor* – a type of browser in which a user can examine variables and methods but not classes. However, a user can change all parts of a class depending on any restrictions the LearningBook author may have imposed as a teacher. Typically any such restrictions are relaxed as the course proceeds. Most importantly the class editor allows the user to edit class-instance variables which could previously only be viewed. This facility is crucial for exploring complex classes, especially when there may be more state on the class side.



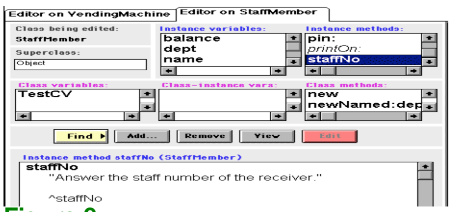
```
new
    "Answer with an initialised instance of the
receiver class; use new from superclass, then
initialise instance."
```

Figure 8

A user can add a class editor page using an **Add page...** menu item from the **Page** navigation button; if this is done the user will be asked to supply a valid class to edit. However the easiest way of using this tool is to use a familiar class browser later versions of which have an

Edit button. This button creates a class editor tool for the class selected in the class browser and adds a page to the current section. For example, the class `StaffMember` might be selected and the **Edit** button clicked to create an editor page for the class called **Editor on StaffMember**, as in Figure 9.

The facilities for changing a class being edited are not very different from those in an ordinary class browsers; they include adding and removing variables, methods and comments. The main difference is that a user can add, remove, view and comment on *class-instance* variables just as you can for class variables or instance variables. A user cannot look up the class hierarchy as you can in a class browser but can see the name of the superclass of the class being edited. Selecting the superclass name (from near the top left of the page) activates the **Edit** button is. One can, therefore, create a new editor for the superclass by selecting the superclass and clicking **Edit**.



Class being edited: StaffMember
Superclass: Object
Instance variables: balance, dept, name
Instance methods: pin:, printOn:, staffNo
Class variables: TestCV
Class-instance vars:
Class methods: new, newNamed:dep

Instance method staffNo (StaffMember)

```
staffNo
    "Answer the staff number of the receiver."

    ^staffNo
```

Figure 9

Class reporter
The *class reporter* is another tool which provides the user with a complete view (a 'report') of a class. In contrast to multi-pane views, this on is a single textual view (which is more like an annotated file-out).When a class reporter is initialized, it selects only the classes defined for the course, all of which are made available in this LearningBook. A user can, if preferred, look only at all the classes visible in the LearningBook by clicking the radio button labelled **all visible classes** (these are defined by the vision for the book and section). Figure 10 shows that this has been done and a `VendingMachine` class has been selected. This report can be printed or saved to a text file, or portions can be copied. Another simple but useful facility is the **Find next text...** button which allows a word or a phrase to be located.

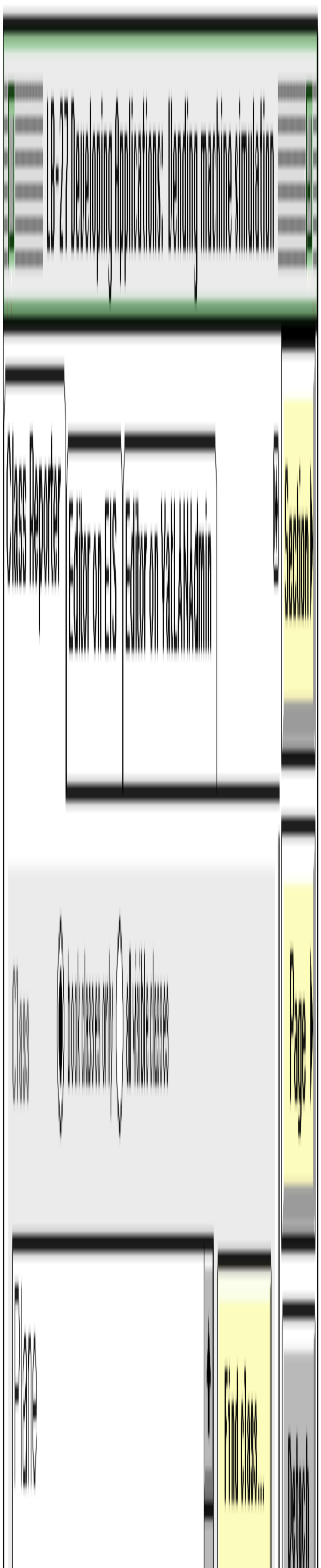
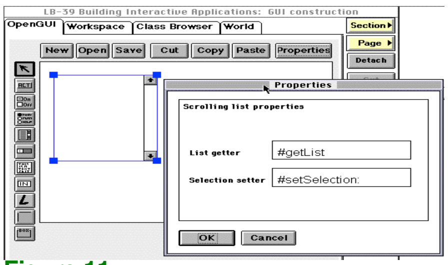


Figure 10

This tool is provided to allow users to read a class as easily as possible and to encourage them to annotate and colour its parts for example within a Notes page (either the one provided or one that they add wherever they like).

GUI Builder
Our requirement to teach a simple separable user interface architecture to students from the outset meant that the code of methods used to move frogs, change their colour or otherwise alter their state, did not contain any code concerning graphical appearance, Instead these methods simply contained `self changed` expressions. Students could easily discover the immediate effect of these messages simply by removing them.

For the notion of separable user interface architectures to mean anything to novices, they have to be able to design their own user interfaces using a GUI builder, and to write all of the model and user interface code necessary to make these interfaces work. Even for experienced programmers using the usual VisualWorks facilities [4] (or any other MVC-type apparatus) could be very daunting. To this end we devised a simplified version of MVC called MUI (Model–User interface) suitable for beginners, and designed and implemented a simple GUI builder, called OpenGUI, which allows students to create user interfaces by direct manipulation of widgets on a drawing canvas, as in Figure 11. For each input and output widget drawn on the canvas, the user must specify via a **Properties** dialog box for the selected widget the appropriate get and set messages which a prospective model must include in its protocol. For input/output fields the type of the message reply of the get method (number or string) must also be specified. When the user interface is complete clicking the **Save** button prompts the student for an appropriate class name for the new user interface.



Scrolling list properties

List getter #getList

Selection setter #setSelection:

Figure 11

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. Note that this mechanism is entirely consistent with the way that a user can add arbitrary pages containing microworlds or tools to a LearningBook; the same dialog box is used, with the course radio button automatically selected. Adding a page is conceptually the same, just that the biding to a model is automatic. And again, section state has been vital in designing a simple and consistent communication mechanism between applications in the same section. After selecting the desired user interface class, an instance is created and inserted as the current page in the LearningBook. The user is then prompted to select an appropriate model from the section dictionary, in other words simply to select a local variable reference. 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 if the right methods are not present in the model, or if they return a message reply of the wrong type. By detaching a workspace from the same section, and placing it side by side with the user interface page, students can view the effect of sending state changing messages to the model from a workspace – just as for microworlds. As long as they have added `self changed` messages to the appropriate set methods in the model, the state changes will of course be reflected in the user interface, thereby reinforcing the notion of separable user interfaces. Similarly changing the state of the model from the user interface can be confirmed by inspecting the model from the workspace.

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 it must understands the required protocol. Similarly, a model from the section dictionary can be associated with any number of user interfaces pages within the same section as the model. All the user interfaces will update the same model and the model will update all the user interfaces, provided of course that the student has added the message expression `self changed` to the appropriate setter methods in the model.

OpenGUI is much simpler than a commercial GUI builder; it supports fewer widgets, and it assumes that a user interface can only deal with one model at a time. 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 software engineering concepts of separable user interface architectures.

5 CONCLUSION

When we started our project in 1994, our overall aim for an environment was that it should support learners in a way that appropriate to the distance mode and, in particular, that it should provide a seamless progression from novice to accomplished practitioner. We have achieved this because of the robustness of the LearningWorks framework and the way in which we have exploited LearningBooks as Smalltalk modules: our clear pedagogy is matched by the design of our environment. This is particularly evident in how we support the principle we call progressive disclosure. What is more we have achieved this on a grand scale, having attracted over five thousand students to learn about object technology and basic principles of software engineering.

During 1998 we deployed four versions of the programming environment, with each successive release providing additional improvements by way of bug fixes, code improvement, and additional behaviour. Two are planned for 1999. During the time most development took place we were using VisualWorks 2 from ObjectShare.. To a large extent the design of OU LearningWorks was influenced by how we were permitted to use VisualWorks classes and, as we have discussed, the size of the Smalltalk image. ObjectShare, has released VisualWorks 3 which utilizes a parcel technology that can make image size significantly smaller. This, and a desire to reconverge with standard LearningWorks will probably lead to further development during 1998–9.

Meanwhile, with colleagues not involved in the design of OU LearningWorks we have begun an objective study of the environment and how learners use it. The project is called An Experimental Student Observatory Project – AESOP [9]. It aims to produce a number of tools for recording and analyzing student interactions with the environment. Among the tools already implemented are a recorder and a replayer which are deployed within our LearningBooks to further customize LearningWorks. The recorder automatically and unobtrusively saves information about significant events while a learner is interacting with a LearningBook. The replayer takes a recording and causes the LearningWorks system to replay the significant events so that an observer (tutor or researcher) can study the learner's actions. A trial is currently underway and tutors who have volunteered are studying student interactions and assisting the research team in its analysis of them. It is likely that a subsequent release of OU LearningWorks will include these tools so as to enable all students to record their work so that tutors could advise them of how they might improve their use of the programming environment.

ACKNOWLEDGMENTS

We wish to thank all our colleagues on the course team and to acknowledge their input to many of the ideas we have reported. A special acknowledgment is due to colleagues who completed the environment: Martin Mathieson, Adam Gawronski and Lucia Rapanotti. We are also indebted to Adele Goldberg, David Leibs and Steve Abel for their help.

REFERENCES

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

[2] Sumner, T. and Taylor, J., The Design of a Personal Learning Manager, *Proceedings CHI '98*, Los Angeles, 1998.

[3] 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.

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

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

[6] Cook, S. & Daniels, J., *Designing Object Systems,* Prentice Hall International, Hemel Hempstead, 1994.

[7] Goldberg, A. Leibs, D. and Abel, S., LearningWorks, *CACM*, 1997.

[8] Goldberg, A. and Ross, J., Is the Smalltalk-80 System for Children?, *Byte*, **6**, No. 8, August 1981.

[9] Thomas, P., Macgregor M., Martin, M., AESOP – An Electronic Student Observatory Project, *Frontiers in Education '98*, November 4–7, 1998, Tempe, Arizona.