Teaching Electronic Publishing: Learning Software Engineering

Richard E. Jones and Ian A. Utting

University of Kent at Canterbury

{rej,iau}@ukc.ac.uk

Abstract

This paper describes an experience of teaching electronic publishing to computer scientists. It takes the novel position of using electronic publishing concepts to support computer science education, rather than the more usual reverse.

Introduction

Courses on electronic publishing (EP) for non-specialists are commonplace in universities. Their scope varies from 'service' courses to those that are an assessed part of undergraduate degree programmes. They may simply introduce students to a particular system (such as *troff* or *WordPerfect*), or may be more extensive, covering document design or surveying available systems and technologies.

Courses for undergraduate computer scientists tend to the latter approach, but must have added weight if they are to maintain their position in increasingly crowded syllabuses. To be useful in anything but the short term, they must balance transferable theory and necessary practice. They should identify and teach fundamental techniques and principles of electronic publishing systems [Brown, 1992b] as well as the principles of use of current ones. The accessibility of many of these principles within EP make it a naturally attractive component of a computer science curriculum, not least because publishing (in the widest sense of the term) is probably the most common computer application, but also because it embodies many more general computer science paradigms. Notions of logical structure, of dynamic and active links (as in hypertexts and active documents) are readily appreciated by computing students. The importance of adherence to standards is also a familiar topic, as are principles of structure and flexibility All these have their software engineering counterparts, with which students will have some familiarity, but never enough.

This paper describes a course given to first year computer scientists that embodies a novel approach. Rather than applying software engineering concepts to EP, it uses ideas of document structure and document design to support the teaching of computer science in a wider context.

Software Engineering

The Joint ACM/IEEE-CS Curriculum Task Force identified three strands within the teaching of computer science: *theory, abstraction* (including modelling) and *design* [ACM, 1991], and defined the core discipline of software engineering as "the systematic application of methods, tools and technical concepts to create complex, software-intensive systems that meet technical, economic and social objectives" [Freeman, 1991], or, more succinctly, as the management of complexity. Practitioners understand that the problems encountered in delivering large systems are not simply scaled up versions of those found in writing small systems, but that by developing methods for controlling complexity, systems beyond the scope of an individual can be built. Designing for change from the outset is a key to the primary goal, that of producing robust and durable software. With modification being the single largest trigger for 'maintenance', post-delivery costs can only be reduced if software is designed for ease of revision [Lehman, 1980].

Abstraction is a powerful technique for managing complexity. Abstract data types (ADTs) are a fundamental design and structuring tool in software engineering, and their use contributes significantly to the reliability and tractability of software. ADTs rigidly enforce the separation of functional specification from implementation. Comprehension of code that uses an ADT requires an understanding only of the behaviour of the ADT, not of its implementation. Conversely, the implementation of an ADT can be modified in the knowledge that this will not affect the rest of the program, provided that the new implementation conforms to the ADT's specification.

The object-oriented paradigm, enjoying an increasing prominence in software engineering education, extends the notion of data abstraction by adding inheritance and object types [Cardelli, 1985]. Code re-use by intention, through the class mechanism rather than through copying, can contribute to productivity, reducing the cost of software creation and maintenance. New classes can be created by inheritance from, and specialisation of, other classes; 'global' changes can be made by modification of an appropriate class in the hierarchy, immediately affecting instantiations of itself and its descendants.

In all areas of computing, these processes appear prominently and indispensably. Unfortunately they are exactly the concepts that novices find the hardest to grasp [Kilgour, 1992]. Demonstrations of their relevance and realistic examples of their use are difficult to reproduce at early stages of the curriculum, precisely because of the question of the scale of the tasks which students can feasibly tackle. Although students apparently accept the rationale for abstraction, the evidence of the code that they write indicates that they do not always take these principles to heart. One reason for this may be a lack of experience in manipulating concrete examples of ADTs. Another might be that, in any conflict between ease of implementation and ease of revision, the former usually wins[1]: students rarely have to maintain their work. In any event, students often seem to view program construction as a simple battle with the compiler and run-time system, with wider issues too easily lost in the smoke.

Electronic publishing

Many of the techniques and principles of software engineering have been applied to the field of EP. Ideas of abstraction are as relevant to EP as they are to computer science [Brailsford, 1988]. Just as programs should be designed with their future maintenance in mind, so documents should be designed for revision. Often documents have lives of ten or twenty years, and thus they share similar life-cycles to software products. Portability and tractability is an issue for both camps. International standards such as ODA [ISO, 1989] and SGML [ISO, 1986] have been developed for the representation and interchange of documents. Both are specifically intended to cater for revisable documents.

Creating and maintaining a *logical* structure for a document has many advantages. The fundamental principle is the separation of the creation of a document from its design. Using programmer's terminology, this principle is one of data abstraction, separating the specification of an object from its implementation (or physical format), and from the data (text) associated with instances of a particular type. A document processing system can take advantage of the logical structure of a document to provide section numbering, cross-references, and so on, automatically because the structure reflects the author's perception of the document. House styles can be imposed and document formats revised with the same ease that ADTs allow the programmer to change the implementation of data structures.

The tools

Computer Science at the University of Kent aims to be a technically challenging, broadly based degree underpinned by a sound theoretical foundation. Most practical work is carried out within the Unix environment and many graduates emerge with a strong bias towards it. It became

---

[1] unless the artificial and external sanction of losing marks is imposed!

apparent that the whole microcomputer revolution had bypassed the Kent undergraduate computer science programme[2], and indeed it was possible to graduate without ever being told what a spreadsheet was.

The restructuring of the degree in October 1991 presented the opportunity to add a new first year option introducing Computer Science students to microcomputer technology and exposing them to some of the more important application areas. The staff responsible for the course wanted students to see and to use state of the art microcomputer software, and to be exposed to the widest possible variety of microcomputer platforms, including the Macintosh and OS/2 as well as DOS and Windows. Given the pre-eminence of the Macintosh in EP at this level, it was an obvious choice of platform. The packages chosen as exemplars were *Microsoft Word* and *Aldus PageMaker* as well as, of course, the Macintosh operating system and graphical user interface itself. This choice also allowed the course team to address an affective goal of this (and many other) courses in the programme, that of improving the students' written communication, and especially in the case of the Macintosh, their more general visual literacy.

The rest of this paper describes the objectives for this EP section of the course, how we set out to achieve them, and how they are used explicitly to address the software engineering ideas introduced above.

Microsoft Word is a fully featured word-processor. Its most interesting feature for the purpose of this exercise is the ability to assign named *styles* to 'paragraphs'[3]. In this context, a paragraph is any sequence of characters or graphics delimited by a paragraph marker (carriage return). A number of pre-defined styles are available and the user can also define new styles, each of which may be based on an existing style. Paragraph styles may include such details as font, leading, inter-paragraph space…

Aldus PageMaker's power allows the user almost complete control over the placement of objects on a page. It also has some limited tools for the creation and modification of text and graphics. Unfortunately, *PageMaker's* flexibility offers the naïve user sufficient rope to hang himself or herself. Fortunately, *PageMaker's* styles provide a discipline within which the publication can be controlled allowing greater consistency and ease of revision of appearance. *PageMaker* can import *Word's,* and other word-processors' styles; alternatively it can read the paragraph *tags* of an ASCII file.

Teaching the principles

The course in question is an assessed part of an honours degree programme: it is not simply a training course. Fundamental techniques and principles of electronic publishing systems are stressed. Some understanding of the background of publishing and its terminology is essential, as is some idea of design and an understanding of good typographic practice [Williams, 1992] (or at least enough to give students sufficient aesthetic sense to discourage them from making unreasonable decisions). But the course team wanted to do more than teach good EP practice.

We wanted to teach the course in such a way that it would support other parts of the degree programme. The communication skills aspect has already been mentioned. Electronic publishing has been described as the use of computer science and electronics for the production, manipulation and dissemination of information [Brailsford, 1989]: the skills-transfer from computer science to EP is well understood. What we hoped to do was to put this process into reverse, to use ideas from EP to reinforce the computer science that our students were learning.

Lessons on the need to control complexity and to design for evolution and portability apply

---

[2] Perversely, Kent students taking multi-disciplinary degrees combining a social science subject with a computing component had more experience of the range and power of microcomputer based applications than did those students taking Computer Science degrees.

[3] but unfortunately not to other character sequences.

equally to large software projects and documents. EP uses structure and abstraction to separate design from content, giving the benefit of easily modifiable house styles. Software engineers use ADTs to separate functional specification from representation, allowing easy revision of implementations of data structures. The relationship between objects of varying degrees of similarity can be captured by *Word's* and *PageMaker's* styles (using their 'based on' capability) in the same way that inheritance can be used in object-oriented languages. We believe that allowing students to *learn* about these concepts in an EP framework has several advantages, especially for novice programmers. Documents are more familiar than computer programs, the effect of changes made to a document in a 'WYSIWYG' environment are more immediate, and larger yet still feasible tasks can be set as exercises. The benefit of modifying a single style rather making numerous manual changes is readily apparent; on the other hand, the latter method is still possible, although tedious and error-prone.

The task

As part of the assessment of their understanding of the course, students were required to undertake an exercise where they played the role of publishing companies whilst the lecturer played that of the customer. The task was to produce the University's Computer Science Degree Handbook, a reasonably complex and substantial document. The customer provided a 'typed' copy of the handbook annotated with a proof-reader's comments. These comments specified fonts, the level of headings, margins, line and paragraph spacing and so on: the specification was deliberately fairly tight. Text-only versions of the handbook were available on the network (it was not intended to be a typing exercise).

The students were asked to produce a copy of the handbook using *Word*. It was made very clear that the customer was notoriously fussy and that they must adhere tightly to the specification. They were also warned that the customer was likely to have changed his mind about the format of the handbook by the time they came to use *PageMaker*. A deadline was set by which the handbook must be completed and handed in electronically. (The precise mechanism for handing in was a 'drop folder' to which they could copy files but not otherwise access them.)

The project then moved to a second phase: the students were asked to produce a final paper version of the handbook with *PageMaker.* This version would incorporate the word-processed text as well as additional graphics, a table of contents…

Analogies

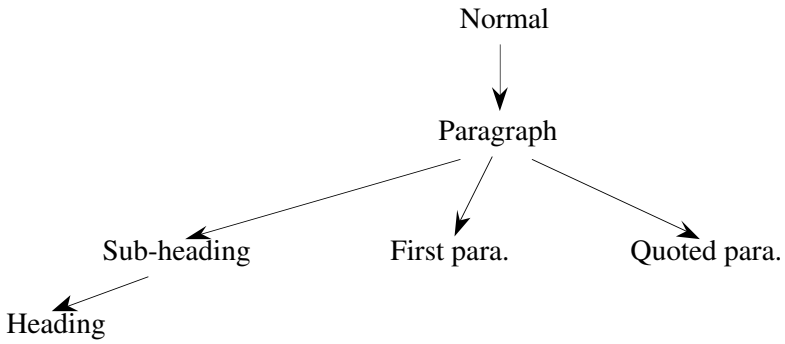
Students were invited to attack the assignment as they might a software project. Prior to the assessment they have at least half a year's experience of structured program design, and have also been introduced to informal specification and ADTs, but do not meet object-oriented design (*per se*) until later in their academic careers. Following completion of the assessment, they were encouraged to draw analogies between their approach to the task and software engineering methodologies. We describe some salient points below.

Specification: The job required students to design from and adhere to a specification. This is always a useful discipline, and experience suggests that it is not one that comes naturally to undergraduates.

Modelling: The first step was to identify all classes of object in the document. Such classes include headings, sub-headings, body text, and so on. This form of problem decomposition is most notably associated with the OO paradigm, but at this level of detail, is equally the approach used in entity-relationship modelling and in ADT-based design.

Inheritance: A logical hierarchy of object classes could now be identified. *Word* and *PageMaker* both allow a paragraph style to be based upon an existing style (single inheritance in object-oriented terminology). Considerable effort can be saved when the format of the document is revised if the style hierarchy reflects the logical hierarchy of the document. This means designing for evolution from the outset. There is a clear analogy between the separation

of document design from creation (electronic publishing) and the separation of functional specification from implementation (ADTs in computer science).



Style hierarchy, showing inheritance

Implementation: Having defined the format of each style, the document can be 'marked up' logically by assigning styles to paragraphs. Procedural markup — explicit formatting changes for individual paragraphs — is discouraged in the same way that, for example, the hard coding of literal constants in programming assignments is discouraged (and for many of the same reasons). The assessment is submitted electronically, which makes it possible to examine students' style hierarchy explicitly and to determine whether styles have been consistently applied, in a way that it is not possible to infer from a printed version [Brown, 1992a].

Revision: Students who adopted this discipline from the outset were rewarded when, in the second phase, revisions were demanded. If their document had been designed with a sufficiently rich style hierarchy, all formatting changes could be accomplished by modification of the appropriate ancestor style. If not, a large number of tedious and error-prone changes would have to be made manually. The analogy with abstract data types is clear. Furthermore, one of the attributes of a *PageMaker* style is whether the paragraph should be included in the table of contents. By making this part of the style of the appropriate level of heading, a table of contents can be created with little further effort. The idea of a style as an example of ADT could have been further enforced by disabling menus in *Word*. By this means, the only way to modify the appearance of a paragraph would be to apply a different style to it (we did not take advantage of this feature).

Results

The course is a popular option and is well-received: typically 80% of the eligible intake opt for it. Students readily grasp *Word* and *PageMaker* and appear to accept and apply ideas of structure and abstraction. Almost all students recognised the need for separation of design from content and designed styles for the assessment accordingly; most then applied their styles to paragraphs conscientiously. However fewer students provided other than fairly flat style hierarchies. This is not particularly surprising: structured design should be fairly well established by this point in the course, but inheritance is not yet a part of the students' programming vocabulary.

It is early days yet — the first cohort is still to graduate — but there is some evidence that our strategy has borne fruit, although the popularity of the course has left only a small control group against which measurements can be made. Nevertheless, in the first year examinations, students who took this course substantially out-performed, on average, those who did not.

Computer Science and EP share many techniques and principles. Unfortunately students often lose sight of software engineering issues — 'programming in the large' — as they struggle with the immediacies of coding in the small. This paper has described a course in EP which offers a way of bringing these principles to the fore. The software engineering issues are present, but lower-level complexities are removed: a paragraph never core dumps! Furthermore, all students are familiar with the notion of document production, in principle if

not in practice. In a WYSIWYG environment the results of their efforts are immediately apparent.

Reinforcement of the ideas of abstraction is particularly useful to these students who do not have a strong mathematics background. Throughout the degree programme, they appear to have more difficulty with abstract ideas than the rest of their cohort. This exercise provided a concrete example of using abstraction and of the benefits which accrue. Although the exercise did not enforce the separation between logical specification and representation, it did offer significant rewards for so doing.

Finally, it is not so clear that the course was a success in terms of design. The range of material that first year computer scientists apparently consider to be aesthetically pleasing is quite remarkable!

References

[ACM, 1991] Report of the ACM/IEEE-CS Joint Curriculum Task Force Computing *Curricula 1991* February, 1991. A summary appears in CACM 34(6), June 1991, 69-84.

[Brailsford, 1989] D.F. Brailsford *Electronic publishing and computer science* Electronic Publishing — Origination, Dissemination and Design, pilot issue, 13–21, 1988.

[Brailsford, 1989] D.F. Brailsford and R.J. Beach *Electronic Publishing — a Journal and its Production* Computer Journal 32(6), 482–493, December 1989, Cambridge University Press.

[Brown, 1992a] P.J. Brown and R.E. Jones *Marking EP coursework using electronic submission* Teaching Electronic Publishing, Proceedings of TEP'92, (ed. J. André), BIGRE 79, 36–45, April 1992.

[Brown, 1992b] H. Brown and I.A. Utting T*eaching electronic publishing to computer scientists* Teaching Electronic Publishing, Proceedings of TEP'92, (ed. J. André), BIGRE 79, 97–105, April 1992.

[Cardelli, 1985] L. Cardelli and P. Wegner *On understanding types, data abstraction, and polymorphism* ACM Computer Surveys, 17(4), 471–522, December 1985.

[Freeman, 1991] P.A. Freeman and M-C. Gaudel *Building a foundation for the Future of Software Engineering* CACM 34(5), May 1991.

[ISO, 1986] ISO 8879 Information Processing — Text and Office Systems — Standard Generalised Markup Language, 1986.

[ISO, 1989] ISO 8613 Information Processing — Text and Office Systems — Office Document Architecture (ODA) and Interchange Format, 1989.

[Kilgour, 1992] A. Kilgour and T. Mayes *Support for the Conceptualisation and Learning of Abstract Data Types* Proceedings of Conference on Developments in the Teaching of Computer Science (eds. D.C. Bateman and T.R. Hopkins), University of Kent at Canterbury, April 1982.

[Lehman, 1980] M.M. Lehman Programs, *Life Cycles and the Laws of Software Evolution* Proceedings of the IEEE, 68(9), 1060–76, 1980.

[Williams, 1992] R. Williams *The Mac is not a typewriter* Peachpit Press, 1992.