Full paper in Proceedings of the 16th Annual NACCQ, Palmerston North, New Zealand, July, 2003 (eds) Mann, S. and Williamson, A. www.naccq.ac.nz

An Example of Teaching Journeyman Level Programming: XML Conversion of Course Descriptors

Dr Mike Lance
School of Computing
Christchurch Polytechnic Institute of Technology
Christchurch, NZ
lancem@cpit.ac.nz

ABSTRACT

This paper evaluates object-oriented design features used in an application that converts course descriptors from text to xml. The discussion of design features of the application is presented as an example of the sort of teaching activity needed to bridge the gap between initial exposure to object oriented programming in a specialized environment and the final journeyman learning experience achieved by a capstone project.

1. INTRODUCTION

I am always surprised when I hear a programmer say that they don't value object technology. I wonder what I'm not "getting" because I find objects a wonderful programming tool. What they have not "got" that they end up wondering what all the fuss is about, rather than saying "wonderful"? I am however one of life's hopelessly eager and non-sceptical adopters of new trends: an avid seeker of 'new fields and fresh horizons'. In contrast are undoubtedly those who have sufficient experience that they see nothing new under the sun and are deeply suspicious of any hyped "silver bullet" like object technology. (Page-Jones, 1998a) However object technology is no longer a new thing but has had time (from the early 1970s) to become a mature technology. Part of the bad press that object technology gets is due to the difficulty of illustrating its benefits to students learning a first programming language. An object-oriented hello world is a very poor teaching example that lacks simplicity and clarity (Westfall, 2001). Educators are in general agreement (Crawford 2002) that ooHelloWorld is not a Good Thing that is "self-evidently wonderful to anyone in a position to notice."(Raymond 1996) In contrast to a simplistic example, this paper discusses the design and implementation of a "righteous solution" (DeGrace & Stahl 1990) to the "wicked" problem of converting legacy system semi-structured documents into XML. Be on the look out for the reasons why a design feature is seen as a Good Thing.

2. THE APPLICATION

The purpose of my application was to convert course descriptors into XML. Course descriptors at CPIT are created in MS-Word and are maintained in a directory as authoritative master documents that are used as the basis for creating ((via cut-and-paste) programme handbooks, course descriptors and outlines, web pages and assorted other advertising materials. There are significant problems with the version control of all these derived documents. There is a need for a system that automatically regenerates all the published documents without relying on manual processes. It would also be valuable to be able to detect which documents need to be republished once a master document is altered. A technical solution which can achieve this is to convert the course descriptors to XML and to use XSLT to extract the published information in multiple formats.

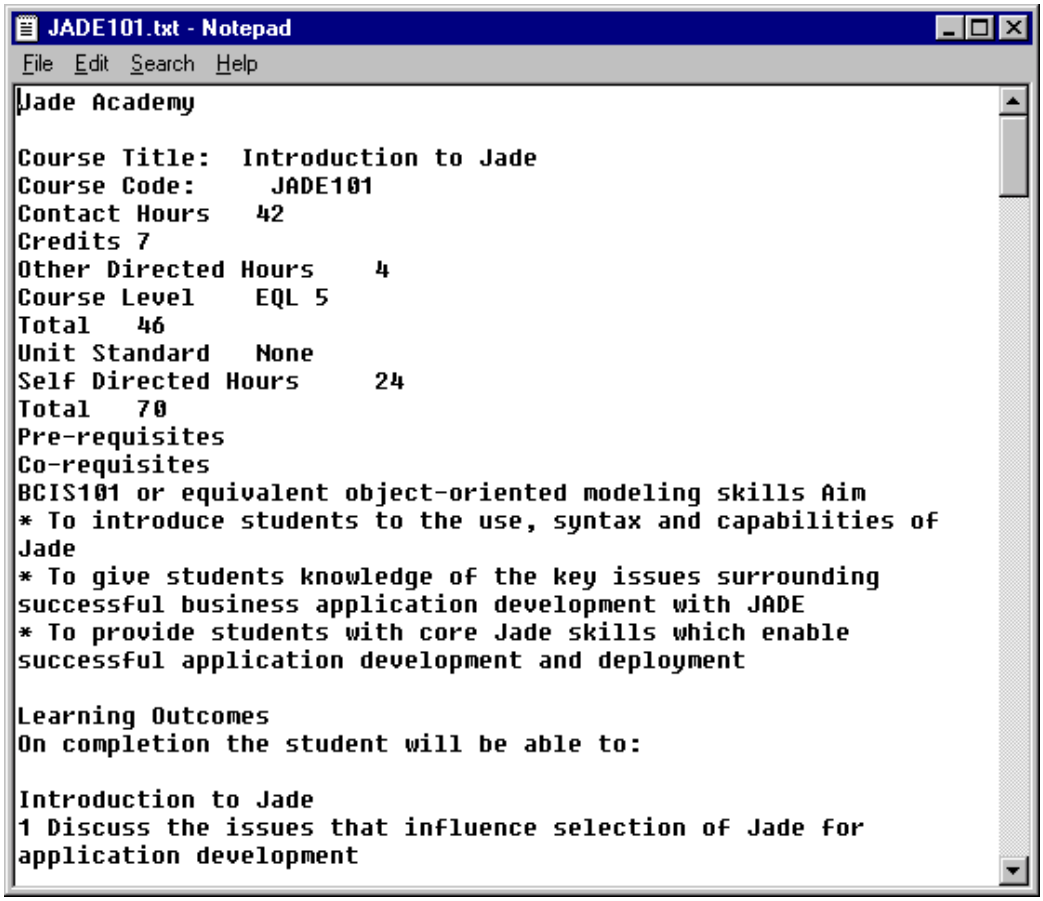

Figure1. This shows the alluringly semi-structured nature of a course descriptor. Note that the header of the Aim section has ended up appended to the end of the prior line. Parsing data with these slight irregularities makes this a wicked problem

An initial attempt at converting from MS-Word into XML by use of a “Save as” option was not a viable solution because the assorted authors of the initial documents had used many different word processing style techniques to achieve a desired format. Also the documents are heavily formatted and one of the goals of a data-centric use of XML is to separate semantic content from display information. Converting a document to XML markup involves extract the meaningful data from amid the formatting information and making sure the extracted information conforms to a standard constraining description. (The ‘XML-speak’ for this is “validation against a document type definition”).

The second attempt at conversion to XML began by saving the course descriptors in text format to get rid of most of the distracting visual formatting information such as changes of font. (See figure 1) Even with most formatting information removed the conversion process was still not simple. Presentation information such as white space, leading stars and numbers still needs to be discarded. Although the sequence of information remains the same regardless of the course or programme and although each area has an almost standard heading, some course descriptors have more information than others and some even introduce new “fields”. Word-processed documents are essentially ‘free form’ and can be endlessly modified without any constraints. The number of the different document formats meant there was a need to detect subtle changes and consequently adjust the rules that search for and extract subsets of information. The need for flexibility makes conversion of course descriptors to XML a Wicked (DeGrace & Stahl) programming task. There were too many documents to make it worthwhile to ‘tweak’ format variations by hand. Besides, creating the application had just got interesting!

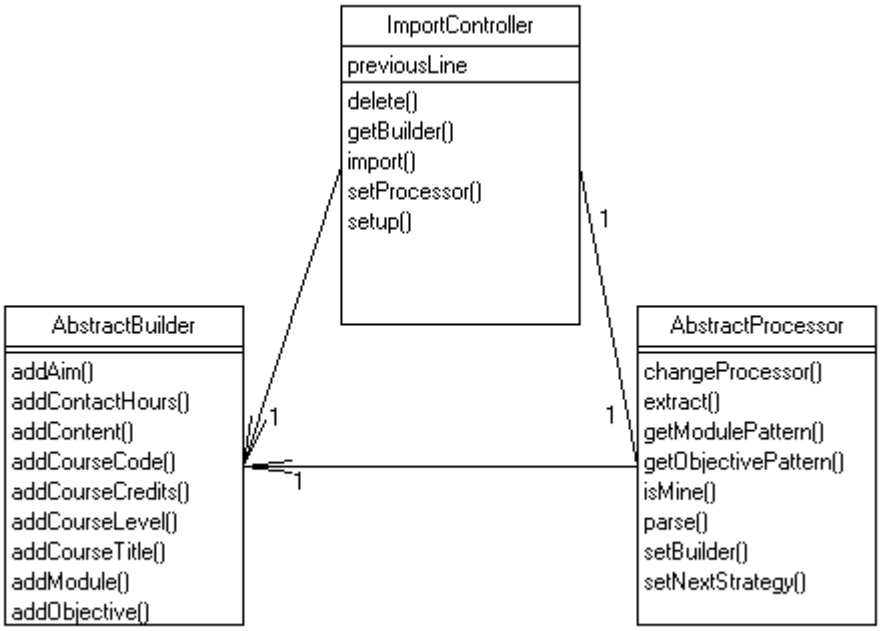

Figure 2. The architecture of the importer application revolves around a controller and two base classes. Many subclasses of AbstractProcessor and AbstractBuilder exist but are not shown here.

2.1 A Decoupled Architecture Allows Component Substitution

There were three stages to processing a course descriptor: accessing and reading in the original document, processing the document to extract the semantic information and outputting the final structured XML document. Accordingly three base classes were created and allocated methods so that between them they defined how the system worked. (See figure 2) The main ImportController class has the responsibility for setting up the object structures of the programme, loading a document and feeding it to AbstractProcessor sub-classes, one line at a time. AbstractProcessor subclasses process each line, identifying and extracting relevant information and then pass it on to AbstractBuilder subclasses to construct the output document. The other main responsibility of the ImportController was to create specialist AbstractProcessors and Abstract Builder subclasses, link them together, and coordinate their operation. The code to set all these structures (see figure 3) is both simple and powerful. Changing the single word denoting which subclass to create after 'as' in the create lines of this setup method results in a different object containing a different type of implementation being 'plugged in'. Abstract classes have been used to define the interfaces how processing code or building code can be called without specifying implementation details. I did this because I knew I wanted to try different ways of extracting information and building the final document. I wanted to flexibility in how I actually coded these processing and building algorithms (Auer 1995, Woolf 2000)

The link between an AbstractProcessor subclass and the ImportController is stored in a reference variable called myCurrentProcessor and each AbstractProcessor subclass has a back-link (inverse reference) named myController. Each AbstractProcessor subclass has a link to an AbstractBuilder subclass named myBuilder. The setup code creates objects and establishes an appropriate object reference by the calls to the setNextStrategy and setBuilder defined in the AbstractProcessor class.

The whole system then works with an implementation mechanism in which "an object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object" (Gamma et al, p360). This can be seen in the import method (figure 5) which is the main loop driving the application. The code reads a line from a file and then if it is not blank passes it to myCurrentProcessor to parse. Exactly which object gets the data to parse is determined at runtime. It can be any AbstractProcessor subclass as they all implement the parse method. This indirect

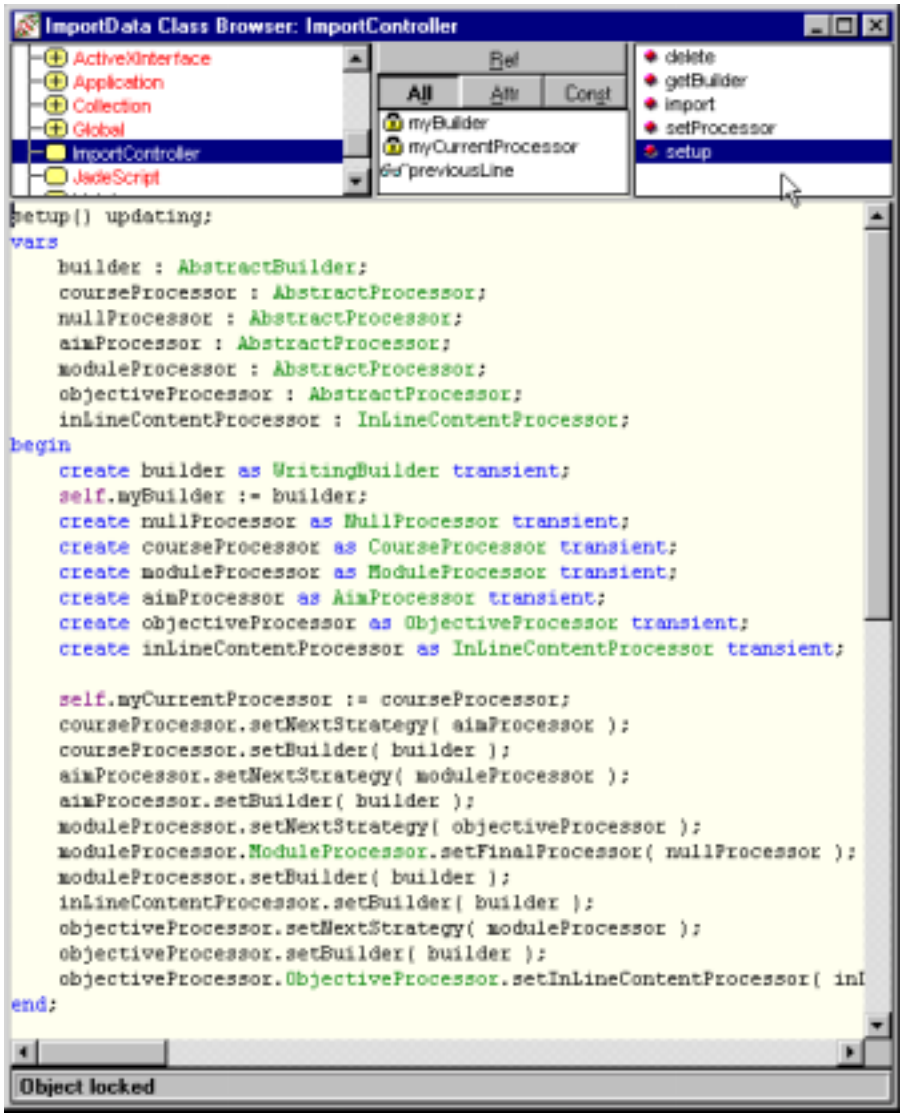

Figure 3. The ImportController creates and links the objects that contain the logic of the application.

referencing and delegation is the feature of object technology that allows components to be substituted for one another.

2.2 A Builder as a Design Class

"Separate the construction of a complex object from its representation so that the same construction process can create different representations." (Gamma *et al,* p97)

The usefulness of delegation can be seen most easily in how a number of different AbstractBuilder subclasses are used. (See figure 5) The base class has a method for adding each type of data being extracted from a course descriptor. Each AbstractProcessor class ends its processing by passing data to myBuilder to deal with. The AbstractBuilder methods are all abstract and do nothing more than promise those subclasses will have to deal with the data. In debugging the application it was useful to firstly create a WritingBuilder class which merely reported the data being fed to it. If all was working well then the course descriptors was reproduced verbatim. A more sophisticated testing version removed the need for visual inspection of data and knew what data to expect and reported exceptions accordingly. A single line of change in a new ImportController would result in data being saved to a Jade database or to an XML data structure. The ability to redirect output and to debug the application without having to deal with the complexity of how to implement the XML processing of the output is a Good Thing.

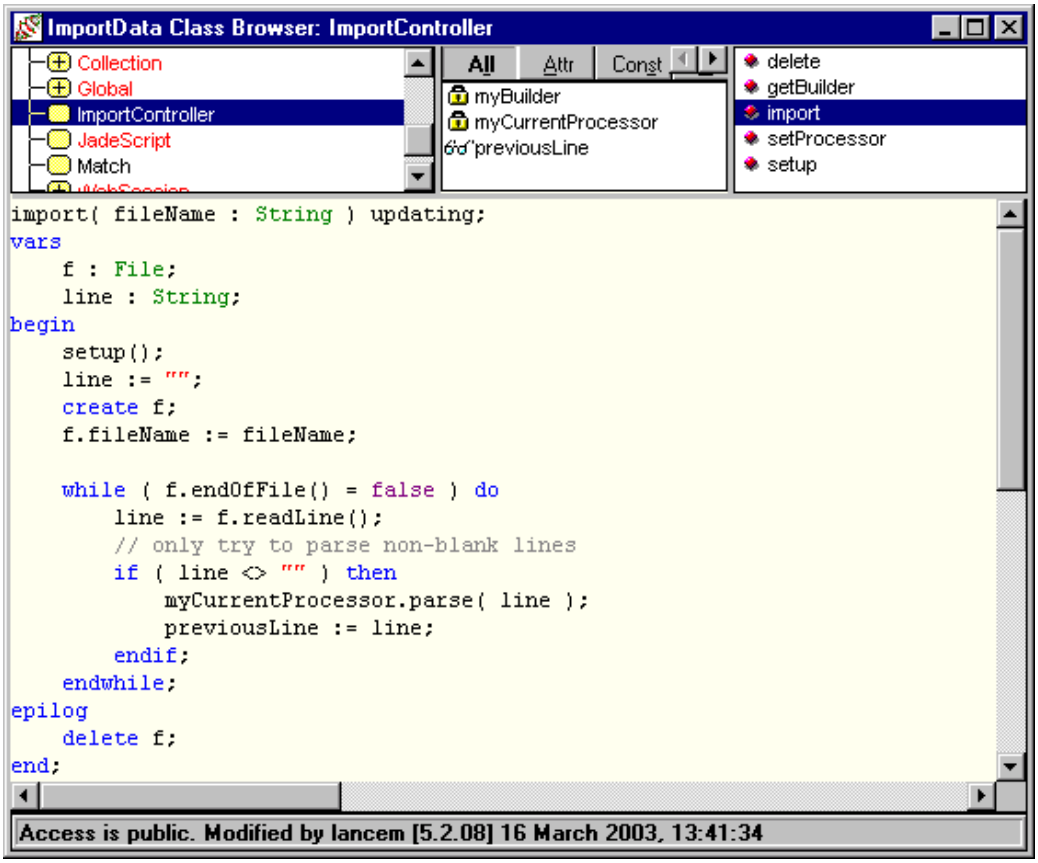

Figure 4. The main functionality of the ImportController is to read a file and have delegated processors parse each line.

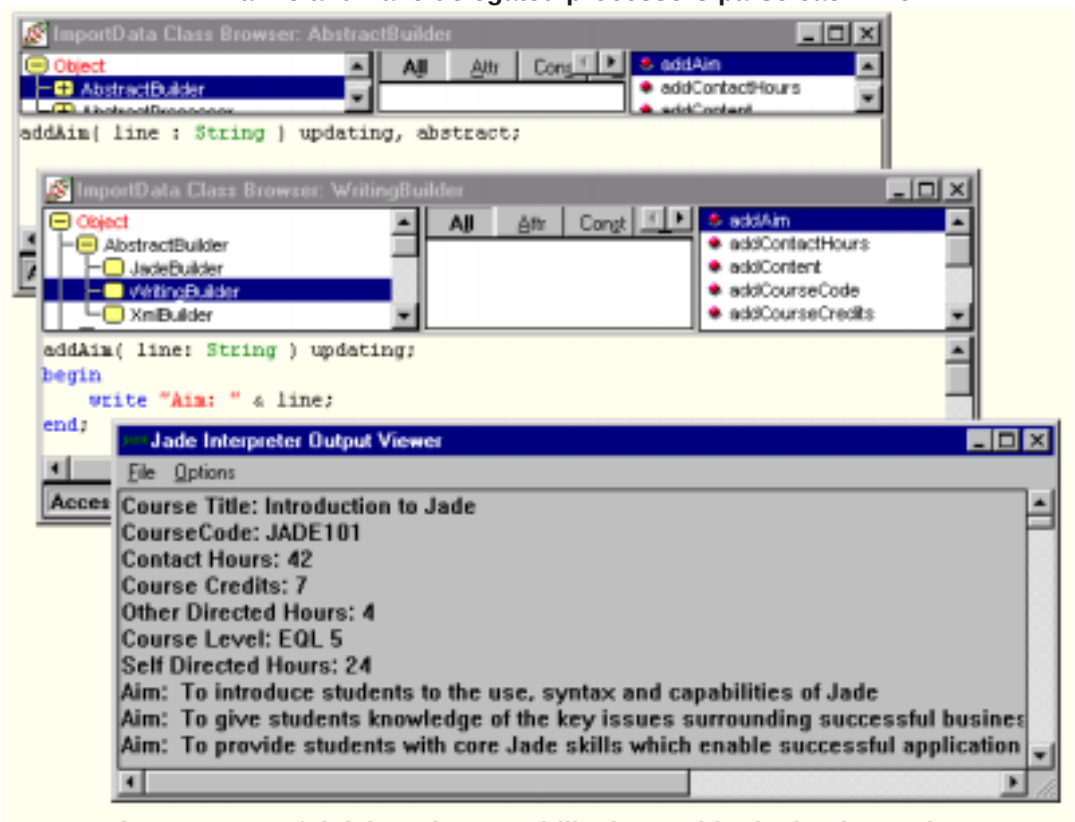

Figure 5. A useful debugging capability is provider by implementing methods in the WritingBuilder subclass.

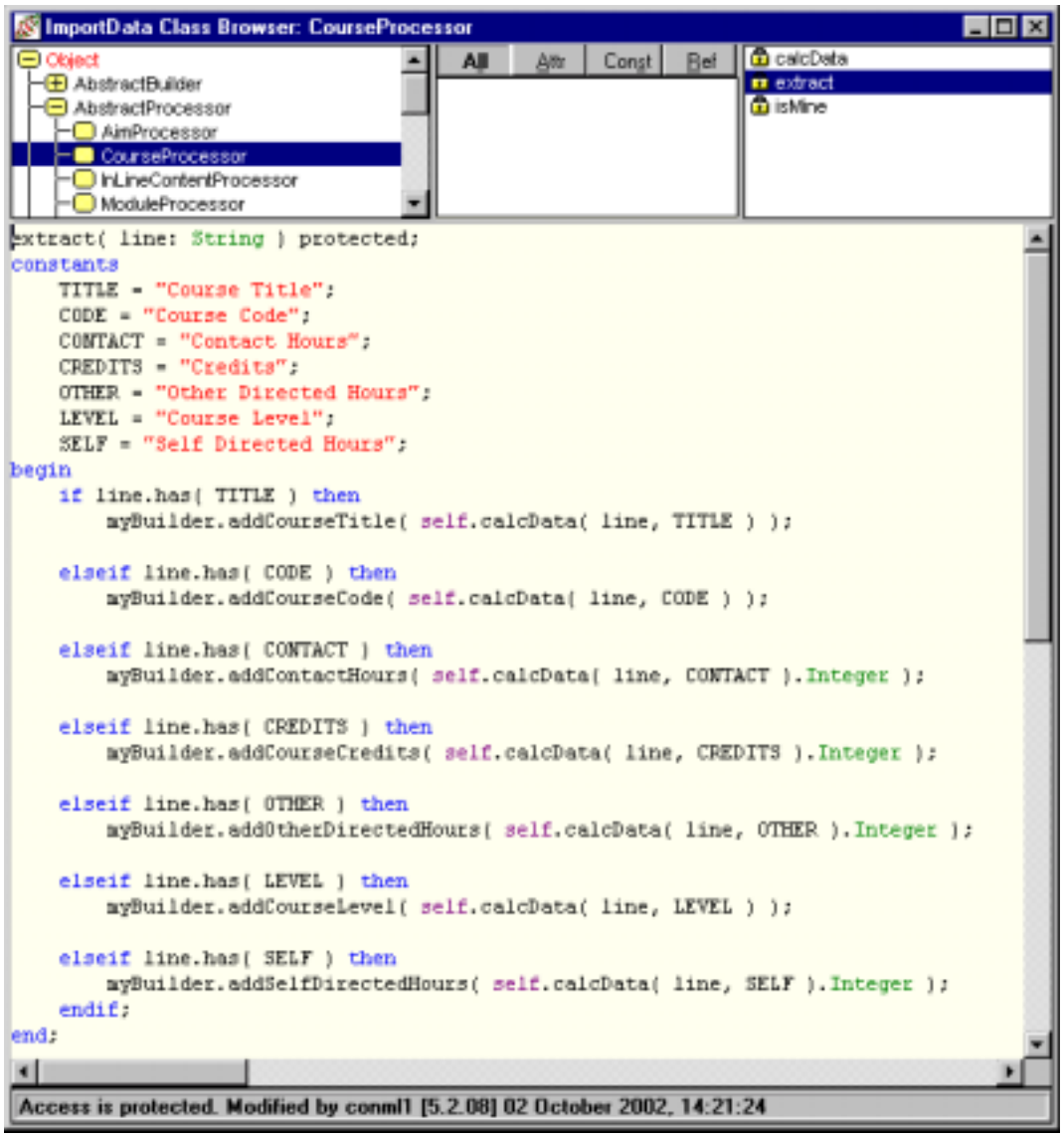

Figure 6. An example of a simple approach to extracting information that ultimately proved to be a too simplistic and inflexible a programming approach.

2.3 The Simplest Thing That Almost Worked

All the decomposing of the system architecture into easily exchanged classes and providing a central control point to flexibly glue the parts of the system together is nice. The AbstractBuilder obviously provides access to a useful debugging technique. However these convenient features do not address the actual problem of extracting information from surrounding presentation details. My first approach to this problem was to try a simple solution. (Wells 1999, Jeffries 2003) This solution involved searching for a pattern that identifies a field of information and then extracting the target text from the line containing the pattern match. (See figure 6.) There are some things to be proud of in this first attempt. The implementation is in Jade and just by using Jade I have been forced to use a strict object based approach: the extract method is contained within a CourseProcessor class. The use of constants at the top of the method provides an easy-to-find place at which to designate the indicator text that determines if a line contains target information. There is of course nothing of object technology involved in the use of constants. Being able to easily find where to make changes is a Good Thing that can be achieved by careful naming and modularization. (Rising 1997) The art here is probably more in the careful naming of things than in any special use of object technology.

There is some encapsulation or information hiding in the code which reduces its apparent complexity to the reader. The .has method is one I defined for all

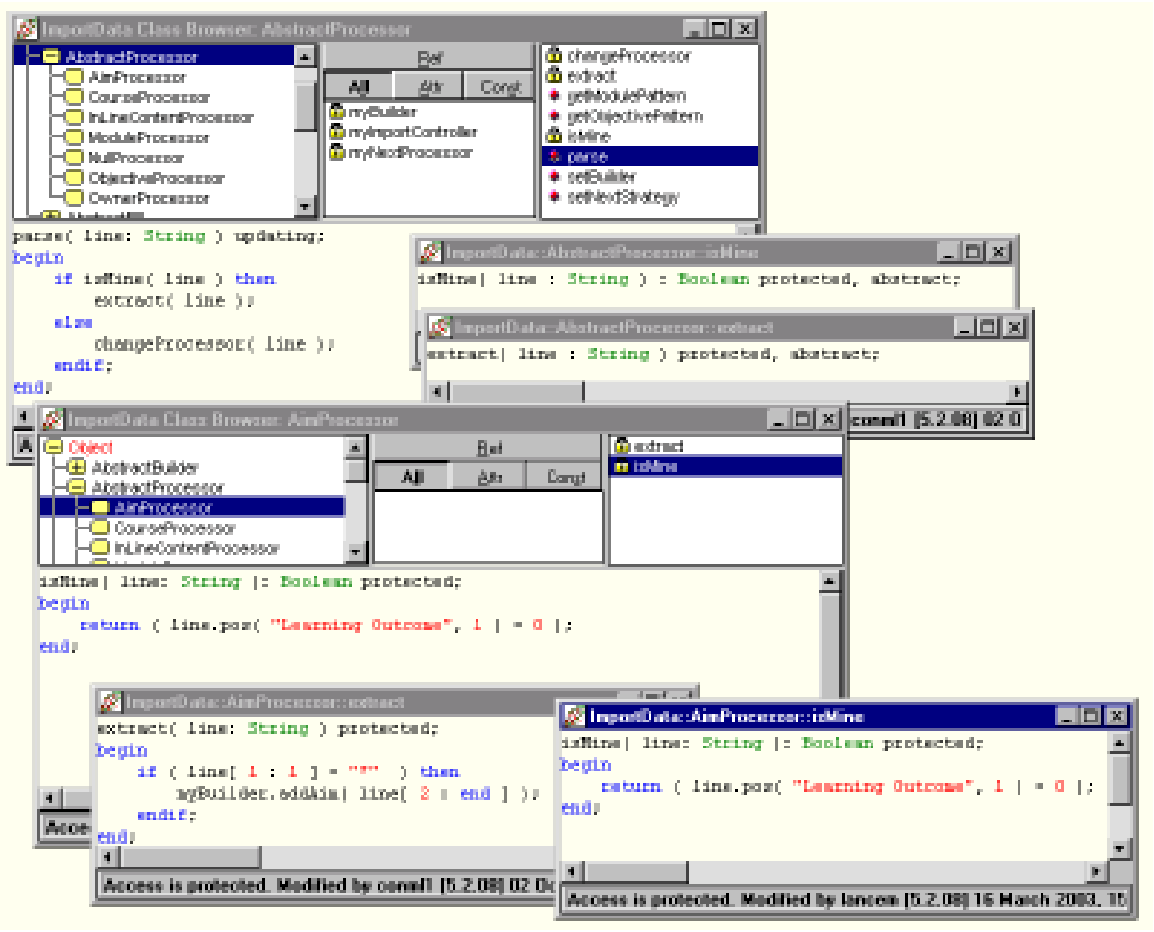

Figure 7. Functional decomposition is enhanced by sub-classing using the Template Method Pattern.

string primitive types: it returns a Boolean value indicating if a substring of the line is equal to a given value. It is good that I can hide away in this method the exact details of substring slicing and indexing because the resulting code is easier to read. The payoff from using an object approach is that all strings contain this method and there is no need for the code to contain extensive data type checking. The method only exists in String objects and thus is guaranteed to be operating on the right type of data. In a similar manner, the .calcData method hides the details of how the target information is extracted from the line, given knowledge of the indicator text. The algorithm here involved removing the indicator text and all white space and punctuation from the line: what remains is hopefully the target text. This is an example of classic functional decomposition and owes nothing to object technology. It does achieve the same useful hiding of complex implementation details.

In trying to continue with the approach taken in CourseProcessor::extract it rapidly became apparent that the code is a very 'fragile'. Extension of the functionality of the method can only be made by editing and expanding the original code. The method requires continual addition of elseif clauses as its functionality is extended to deal with new cases. The simple logic of 'if has then calcData' can rapidly become a complicated series of additional 'or'and 'and' conditions.

2.4 A Template Method extends functional decomposition

"Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." (Gamma *et al*, p325)

The simplest thing THAT WORKED in extracting the desired data was much more complicated that the simplest thing I tried to make work. The AbstractProcessor base class defines its controlling logic in as two template methods (see figure 7)called parse and changeprocessor. The exact details of how

the extract and isMine methods work was left for AbstractProcessor subclasses to define.

For example, an AimProcessor subclass has to extract aims in lines which begin with stars followed by spaces. All the rest of the text on the line is the course aim. The Aims section ends when the words "Learning Outcomes" are detected. The short cooperating methods are a feature of object-oriented programming. They all have a set and focused responsibility. Often they are coordinated by higher level template methods or a controller. This approach is an example of the Open-Closed Principle (Martin 1969) whereby a programme is open for extension in sub-classes classes, but closed to editing of code in the abstract base classes. For example, if, in a differently formatted course descriptor, aims were numbered with single digits rather than stars, then this change is deal with by sub-classing the AimProcessor and only changing the code in the new subclass's extract method. Every new type of course descriptor being processed needs a distinct controller which attaches the appropriate class of processor, but only within the ImportController does any cut and paste extension of code occur. The beauty of this approach is that it maximises reuse of code while avoiding unintended maintenance effects. The loose coupling of the sequence of Processors means that this swapping of processing strategy can also happen at run-time

2.4 Strategy Pattern

> *"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it"* (Gamma *et al.*, p315)

The strategy pattern was used in setting up the AbstractProcessor subclasses. Each AbstractProcessor subclass was passed a line and was responsible for determining if the line held data appropriate to that processor. (See figure 7) If relevant features were detected in the data then desired information was extracted. If the processor could not deal with the data then the ImportController was alerted that the next processor in line was to be tried. The data was passed to the next processor. This effectively allows the processing strategy to be changed dynamically at run time. This chaining of processing strategies avoids the inflexible and fragile hard coding of a logical succession of processing options as was seen in the extract method of the CourseProcessor. Because the successor strategy is stored as a reference, there is the added flexibility of being able to change the sequence of processing strategies dynamically at runtime. This is avoided by the set up code in the ImportController which makes an initial choice of what processing strategy will follow which.

3. CONCLUSION

Introductory teaching of object technology is a problem because "object technology did not improve the ability to design data structures or algorithms, but rather the ability to build complex systems." (Goldberg, 2002, p11) Object technology was developed as an attempt to deal with issues of modularity, reuse and robustness of design under change pressure. These are not the typical concerns of a beginning programmer and the overhead imposed by object technology distracts from the lessons a beginning programmer needs to learn. (Westfall 2001, Crawford 2002). One solution to the overhead required by objects is to develop a custom programming environment for teaching object-orientation based on a careful analysis of the learning needs of beginning programmers. (eg Goldberg et al 1997, Howell 2003, Kolling & Rosenberg 2002) These frameworks, micro-worlds and custom Integrated Development Environments reduce the amount of scaffolding code a beginning programmer needs to write to productively use object technology. There is a recognised progression in the way people absorb sophisticated new techniques and then apply them. (Page-Jones 1998b) The use of a specialist programming environment is widely reported as being successful at moving a student's level of software engineering expertise from "Innocent" through "Exposed" to "Apprentice". However the next step in expertise is to the level of "Practitioner" and is best achieved with the rite of passage of a significant project. Acknowledgement of the importance of this part of the learning experience is shown in the widely adopted capstone project in NACCQ qualifications.

I am concerned that there is not enough emphasis on the length of time a student spends at the "Apprentice" level and not enough emphasis on the sort of skills an Apprentice needs to learn. Skills such as refactoring and use of design patterns are easily left to be tacitly learned by the student programmer. As a tertiary teacher and programmer I have somehow moved myself beyond Practitioner to become first a self-sufficient "Journeyman", then a "Master" who knows when to break the rules and now a "Researcher" looking for flaws in contemporary techniques and consequent ways to improve the techniques. How can I teach my students some of the hard lessons I have learned on this journey so as to shorten their learning pathway? I use a walk-though of a software design, as was presented in this article, to promote the 'use of advanced programming techniques'. I also put strong

emphasis on the development of a cynical or critical appreciation by the student of when object technology is an inappropriate overkill. (Dodani 1999, Kerievsky 2002). Was the design of my application really a Good Thing?

REFERENCES

Auer, K. (1995) "Reusability Through Self-Encapsulation" in Coplien, J.O. & Schmidt, D.C. (Eds) Pattern Languages of Program Design,. Addison-Wesley (Software Patterns Series). pp505-520.

Crawford, D. (ed) (2002) "Forum:`Hello, World' Gets Mixed Greetings" Communications of the ACM, 45(2):11-17

Dahl, O & Nygaard, K (1966) "SIMULA: an ALGOL-based simulation language". Communications of the ACM, 9(9)671-678.

DeGrace, P. & Stahl, H. (1990) "Wicked Problems, Righteous Solutions: A Catalog of Modern Software Engineering Paradigms", Prentice-Hall/YOURDON Press.

Dodani, M. (1999) "Rules Are for Fools, Patterns Are for Cool Fools", Journal of Object Oriented Programming, 6(12):21-23.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) "Design Patterns: Elements of reusable object-oriented Software". Reading: Addison-Wesley Publishing Company.

Goldberg, A., Abell, S.T. & Leibs, D (1997) "The LearningWorks Development and Delivery Frameworks". Communications of the ACM October 1997, 40 (10):78-81.

Goldberg, A. (2002) "Learning is a Community Experience", Journal of Object Technology, 1(2):7-20.

Greenberg, A. C. & Black, A.P. (2001) "Squeak Smalltalk: Language Reference", Accessed March 14, 2003. <http://www.mucow.com/squeak-qref.html>

Howell, S. PyKarel <http://pykarel.sourceforge.net/> , Accessed March 14, 2003.

Jeffries, R. (2003) "Do The Simplest Thing That Could Possibly Work". Accessed March 14, 2003. <c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork>

Kerievsky, J. (2002) "Stop Over Engineering" Software Development, April 2002

Kölling, M. and Rosenberg, J., (2002) "BlueJ - The Hitch-Hikers Guide to Object Orientation", The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark, Technical Report 2002, No 2.

Larman, C. (2001) "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)". New Jersey: Prentice Hall.

Martin, R.C. (1996) "The Open Closed Principle." Accessed March 14, 2003 <http://www.objectmentor.com/resources/articles/ocp.pdf>

Meyers, B. (1997) "Object-Oriented Software Construction, Second Edition". New Jersey: Prentice Hall.

Page-Jones, M. (1998a) "Object Orientation: The Importance of Being Earnest". Accessed March 14, 2003. <http://www.waysys.com/ws_content_al_ibe.html>

Page-Jones, M. (1998b) "The Seven Stages of Expertise in Software Engineering". Accessed March 14, 2003. <http://www.waysys.com/ws_content_al_sse.html>

Raymond, E. (1996) "The New Hacker's Dictionary - 3rd Edition". MIT Press.

Rising, L. (1997) "The Road, Christopher Alexander, and Good Software Design", Object Magazine, 7(1):46-50.

Wells, D. (1999) "Simplicity is the Key". Accessed March 14, 2003. <www.extremeprogramming.org/rules/simple.html>

Westfall, R. (2001) "Technical opinion: Hello, world considered harmful" Communications of the ACM, 44(10):129-130.

Woolf, B. (2000) "The Abstract Class Pattern" in Harrison, N., Foote, B. & Rohnert, H. (Eds) Pattern Languages of Program Design 4, Addison-Wesley (Software Patterns Series):5-14.