South African
Computer
Journal
Number 20
December 1997

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 20
Desember 1997

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarverenging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof Lucas Introna
Department of Informatics
University of Pretoria
Hatfield 0083
lintrona@econ.up.ac.za

Production Editor
Dr Riël Smit
Mosaic Software (Pty) Ltd
P.O.Box 23906
Claremont 7735
gds@mosaic.co.za

World-Wide Web: http://www.mosaic.co.za/sacj/

Subscriptions

	Annual	Single copy
Southern Africa:	R50,00	R25,00
Elsewhere:	$30,00	$15,00

An additional $15 per year is charged for airmail outside Southern Africa

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

Editorial Notes

For two reasons, this edition of SACJ is far later than it ought to have been. The first reason is that there have been some personnel changes in the editorial team. Lucas Introna, having continued for some time as IS editor after transferring to London, asked to be relieved of his duties. Niek du Plooy has kindly agreed to fulfill this role in a temporary capacity until a suitable replacement for Lucas can be found. Due to work pressure, Riël Smit has also withdrawn as production editor, and has been replaced by John Botha. SACJ owes the two retired members a huge debt of gratitude. During his period of tenure, Lucas did sterling work in setting and maintaining a solid standard for IS contributions. Riël put SACJ on a LaTeX path, and has laboured diligently to produce an aesthetically pleasing product. Thanks are also due to Niek and John for their willingness to take over in their respective roles. Until further notice, IS contributors may forward their submissions directly to Niek at his address given on the front inside cover. I shall put successful authors in touch with John for further instructions regarding final preparation of their manuscripts.

The second reason for a delay in this edition has to do with authors who have not scrupulously followed guidelines for producing their final submissions. There have been a variety of problems ranging from missing citations and inappropriate production of figures to incompatible electronic file submissions. All of this, coupled with our new production editor (who—despite an extremely busy schedule—has valiantly climbed a steep LaTeX learning curve) has resulted in an edition that should have been out to press several weeks earlier.

The editorial team will be giving attention to the general matter of format and submission procedures in future. SACJ's citation and reference methods are somewhat archaic and will probably be revised. All the necessary information will be provided on the new SACJ web site at `www.cs.up.ac.za/sacj/`. The site will also contain abstracts of articles in this and future editions.

These are times of conflicting stresses on both the academic and industrial IT communities. They are being felt somewhat more acutely in Southern Africa (and presumably in other developing countries) than in the developed world. Internationally there is tendency to cut back on state financing of universities and a seemingly insatiable demand for IT graduates. Many companies snap up new graduates at attractive salaries, positively discouraging full-time postgraduate studies. International recruitment agencies scour the South African scene for qualified candidates, luring some of our most promising young professionals out of the country. Job-hopping, a drift from academia to industry and from local industry to USA or European industry seems to be the order of the day. Despite the availability of private colleges and institutes, virtual or otherwise, there is a rush of students to university and technikon IT departments, all hoping to get at the IT honey-pot. University administrations are struggling to correct the structural deficiencies of the past and to provide IT departments with sufficient resources to cope with demand. As editor of SACJ, I have no particular competence authoritatively to sum up or analyze these tendencies, but it does seem to me desirable that someone ought to do so. Bodies such as SAICSIT, the CSSA, university authorities, IT industry and state representatives ought actively to pursue joint strategies to ensure that our IT departments are properly resourced and that (non-Zuma) measures are taken to retain graduates in the country. It seems almost redundant to attempt to spell out the consequences of inactivity.

Derrick Kourie
EDITOR

The Abstraction-First Approach to Encouraging Reuse

Philip Machanick
Computer Science Department, University of the Witwatersrand,
2050 Wits, South Africa
philip@cs.wits.ac.za http://www.cs.wits.ac.za/~philip/

Abstract

Experience in industry suggests that reuse does not happen without retraining, despite the fact that many experienced software engineers accept that reuse ought to be more efficient than coding from scratch. A possible cause of the problem is that traditional computer science courses do not emphasize reuse, but teach skills such as data structures and algorithms in a bottom-up way. Since reuse is meant to simplify programming, this paper argues the case for re-ordering a traditional data structures and algorithms course, using an object-oriented language, so that it starts from abstraction and reuse, and postpones coding from scratch as far as possible. The intention is that reuse should be learnt before other strategies, so coding from scratch does not have to be unlearned before reuse seems natural. The paper presents experience with a restructured abstraction-first course, and proposes that an essential tool for such a strategy is a set of scaled-down libraries and frameworks, designed for teaching. Compared with an earlier C++-based course in which concepts were presented in a different order, more ground was covered, without a major change in the students' results.
Keywords: *object-oriented programming, abstraction, reuse, computer science education*
Computing Reviews Categories: *D.1.5, D.3.3, D.2, K.3.2*

1 Introduction

It is widely recognized that practicing reuse does not happen automatically; some form of re-education is necessary, if programmers have been schooled in more traditional coding styles. This effect is observed even when software engineers accept that reuse is more efficient than coding from scratch [3; 4; 10; 12].

So why is reuse hard? Isn't object-oriented programming, complete with reusable class libraries and frameworks meant to make programming *easier*?

Brad Cox [8] has gone as far as to propose that almost all programming should be based on reusable object-oriented components.

Is component software to be restricted to user-level components, as in OpenDoc [2], or is Cox right?

Could the problem be in the difficulty in unlearning previously taught non-reuse strategies? If so, shound't reuse be the starting point for learning programming concepts, rather than an afterthought at best: something learnt in the workplace, or possibly relegated to a software engineering course later in the curriculum?

Order of learning

Some lessons can be learnt from the early years of structured programming and strong typing. Early Pascal texts for example tended to start with the parts of the language that most resembled FORTRAN, and gradually worked their way around to the "new" features—named types, local variables, procedures and functions with parameters, the use of procedural abstraction to reduce the complexity of code. Gradually, authors came round to the view that procedural abstraction (and to a less extent data abstraction) could and should be developed from the start: procedures and functions moved to Chapter 1, user-defined types appeared early, and monolithic main programs disappeared.

Although object-oriented programming isn't a new concept, its acceptance in the mainstream is still relatively new. Consequently, problems of presentation order in course materials are still likely to be an issue. Thought needs to be put into how to add object-oriented concepts into the curriculum, so graduates will be able to place object-oriented concepts into context in the workplace.

Many C++ books, for example, do not introduce code reuse at all; it's common practice for abstraction mechanisms (classes, templates) to be either introduced late, or purely as ways to implement classes from scratch [7; 9; 11; 16; 18; 20; 22; 24; 25]. Even those who do take a more object-oriented approach tend to introduce implementations of their class libraries early [6].

If reuse really simplifies coding, shouldn't reuse be at the *start* of any course on object-oriented concepts, with skills such as design and coding from scratch only introduced later? Is leaving these concepts to late in the curriculum, despite the paradigm shift to object-oriented programming, occurring for the same reason as the persistence of a FORTRAN-like mindset in early Pascal books? Perhaps educators find it hard to believe that it is easy or natural to learn concepts in a different order to the order in which they personally encountered the concepts.

The abstraction-first approach

This paper presents an alternative ordering—called *abstraction-first*—of concepts in an object-oriented revision of a traditional data structures and algorithms course (presented in the second year of a three-year BSc programme), which aims to develop a reuse habit *before* other styles of programming are developed. Furthermore, the general approach of the course aims to draw on important software engineering principles in the way the course is designed. For instance, information hiding is employed: language details which are not needed at early stages are not presented until they are needed.

By contrast, the other common approaches—top-down or bottom-up—differ from abstraction-first in where they first tackle implementation from scratch. The abstraction-first approach is closer in spirit to top-down, in that detail is developed later. However, unlike with the top-down approach, the abstraction-first approach starts with emphasis of reusing other people's abstractions, rather than developing your own.

Data abstraction and algorithms (DAA)

A new object-oriented course using the abstraction-first approach was developed in the Computer Science Department at the University of the Witwatersrand in 1994, with several aims in mind:

- a review of published IEEE/ACM curricula [1] showed that some of our courses should be revised to keep in line with international trends
- the department was introducing a Software Engineering course in the third year of the BSc, and it was considered important that there be some linkages across years
- changes in industry demands for skills in object-oriented languages and design methods required that we introduce object-oriented concepts more strongly (they were previously introduced as part of a third year Programming Languages course)

As with all of our courses, the new object-oriented course, which was called Data Abstraction and Algorithms (DAA), was designed to mesh as far as possible with the rest of the curriculum, so difficult concepts could be treated in more detail each year. For example, algorithm analysis is handled in each year of the degree, and in DAA, the emphasis is on developing the major conceptual tools for algorithm analysis, including the most important proof techniques. Software engineering is also handled in each of three years: in the first year, social and ethical issues are introduced; the second-year DAA course introduces the general concepts of the life cycle without going into detail, and the third year Software Engineering course examines the life cycle in more detail, especially the management issues.

Although a simplistic view of meeting industry demands would have been to adopt the most popular object-oriented language, we chose to focus on concepts, with choice of language dictated more by concerns such as availability of good affordable tools. C++ was the language which most suited the goals of cost and suitable tools at the time, though the language presents some problems with focussing on object-oriented principles.

The course which DAA replaced, Advanced Programming, was presented in Modula-2, and covered much the same ground in terms of data structures and algorithms. However, the time for DAA increased from 7 weeks to 10 weeks to allow space to introduce Software Engineering. We felt it important to explain how concepts like object-oriented programming and algorithm analysis fitted into the broader scheme of things.

Focus of paper: DAA and abstraction first

The discussion in this paper is not however based on transition from Modula-2 to C++, but on improvements in the C++ course, since the change from Modula-2 to C++ introduced too many variables (change in curriculum, change in total time) to make for easy measurement of the effect of the change. The findings presented in this paper relate to differences between the final C++ course and an initial attempt which did not use the abstraction-first approach.

The emphasis in this paper is on the effect of changing to a more reuse oriented strategy, the abstraction-first ordering. Just as later structured programming or Pascal texts [23; 13; 17] started with procedures, user-defined types, and so on, this paper argues that object-oriented texts should be ordered in the way proposed in this paper. Not only is learning reuse early beneficial in introducing an increasingly important aspect of real-world software engineering practice into the classroom, but if reuse and more particularly abstraction are truly valuable tools, a reuse-oriented approach could potentially make learning easier. Furthermore, basing the course on reuse makes it possible to structure introduction of the language in such a way that more difficult constructs such as loops are introduced later. Even for students with prior exposure to programming, deferring difficult concepts may be of benefit.

The abstraction-first ordering is pursued as follows in the DAA course. Students are initially introduced to the idea of abstraction, starting from real-world examples (e.g., how a bus can be viewed at different levels of abstraction), and then a virtual machine such as a good user interface design (the Apple Macintosh is used as an example, since that is the equipment used for DAA). The next step is to introduce object-oriented code as a way of building programs from building blocks, rather than from scratch. The next level of complication is the idea of incomplete types, in the form of templates—which are introduced through a library of container classes. Once the essential idea of classes and objects is established, libraries and frameworks can be introduced. Only after the major concepts are in place is programming in C++ introduced, starting from programming almost entirely using reusable components.

Once this background is built up, it becomes possible to branch out to design decisions: object-oriented design is introduced (with emphasis on working from a design), along with test strategies, algorithm analysis and methods of choosing data structures. By the time detailed algorithm analysis has been introduced, the students have been thoroughly exposed to the reuse ideology, and algorithm analysis is shown to be important in choosing the right reusable code, before its use in designing algorithms from scratch. To fit in with the intent of contextualising the course in the bigger picture of software engineering, algorithm analysis and data structure analysis are presented in the context of working out a design—though design methods and further detail of the software life cycle are deferred to the third-year course on Software Engineering.

Towards the end of the course, algorithm analysis becomes a springboard for presenting principles of designing containers, as well as classic sorts and searches. In other words, the final and most difficult object-oriented skill taught in the course is that of designing general, reusable components. By the end of the course, the students should have learnt that the default programming strategy is to reuse—but they should have the concepts to start from scratch if need be. Better still, since they should be thoroughly schooled in reuse, if they do code from scratch, they are more likely to think in terms of good abstractions that can be reused. Once they go into the third-year Software Engineering course, they should be in a position to relate principles such as reuse and algorithm analysis to the software life cycle.

At early stages of the course, relatively little detailed coding is required of the students—but availability of a good selection of reusable classes and templates is essential. Toy libraries and frameworks are useful educational tools in such a course—just as a toy machine can be a useful tool for teaching computer architecture, or a toy language can be useful for teaching programming. Common commercial libraries and frameworks (such as PowerPlant and MacApp on the Macintosh, or Microsoft Foundation Classes on Windows) are too large and complex to use to teach principles. If such a library were to be used, much of the course would be taken up in learning the library.

Short-term evaluation of the DAA course shows that the class has responded well to the course, in that their results are comparable with those of the previous year, which introduced C++ in a more conventional fashion, despite more material being covered. Evaluation should ideally take place through work place studies, after students who have been schooled in the abstraction-first approach have attempted to work on a real project, where their ability to practice reuse can be evaluated.

Organization of the paper

The remainder of the paper is organized as follows. The next section contains a description of the design decisions for the new course, based on the abstraction-first principle. Next comes an outline of the object-oriented data abstraction and algorithms course, with an explanation of how the order of presentation changed between the first version of the course and the abstraction-first version. The following section discusses ways C++ inhibits implementation of an abstraction-first ordering. The paper then discusses how introducing the abstraction-first approach made it possible to cover more ground than with a more conventionally ordered C++-based course, which was presented the year before.

In conclusion, the paper weighs up the new course against the pervious version of the DAA course, proposes future work and outlines a strategy for future authors of object-oriented texts.

2 Abstraction by Design

The fundamental principle which has been used throughout the design of the new DAA course is to expose the students to just as much detail as they need to understand a specific concept, but no more. The idea is that students cannot be expected to see the point of abstraction, if the course itself dumps them into detail indiscriminately. In this sense the course itself is based on the information hiding principle.

This design principle has driven the ordering of chapters, and of sections within chapters.

First, to appreciate abstraction, it's not necessary to know about programming—so real-world examples are used to illustrate abstraction before computers are mentioned at all. One example which is used is that of a farmer who is persuaded to trade his tractor in for the latest, best new model in the show room, which has a very good sound system but isn't very good at pulling a plough. His mistake? He was dazzled by details and forgot what his major purpose for his old tractor was.

Then, to introduce abstraction in programming, the course shows that it's possible to reuse code without knowing anything about how it's implemented. Most members of my class have no prior exposure to C or C++, so keeping them ignorant of implementation is a simple matter. C++ syntax is introduced from the header file inwards to the compilable file. In this way, the students are more familiar at first with interfaces than implementations. Booch diagrams [5] are introduced early, so they learn that object-oriented design is not language-specific—they learn to think of classes and objects before they know enough C++ to be locked into its peculiarities.

When detailed implementation is first introduced, it is introduced in the context of implementing a small, separately compiled part of a partially constructed program, so as to demonstrate the value of abstraction as a tool to shield the programmer from the complexity of a larger program. The larger example is based on a simple application framework, MiniApplication with a fair amount of predefined behaviour. It's possible to use

MiniApplication to build a simple graphics editor with very little additional code.

When the course finally moves to larger examples, the strategies of reuse and hiding other parts of a larger program are maintained as far as possible. When object-oriented design is introduced, part of a design is presented, and it is demonstrated that a small part can be implemented, only knowing the interfaces to the rest of the design. Only when the general idea of abstraction is well established does the course venture into algorithm analysis—and even there, algorithm analysis is interleaved with a section on design for reuse, so the point is not lost: you only code from scratch with reuse in mind.

3 Implementation Detail versus Original Attempt

Educators schooled in the structured programming-derived order of presentation are generally skeptical of the abstraction-first approach when it is explained to them. They ask how non-trivial examples can be presented without explaining major aspects of language syntax, such as loops. The point they make is a good one: students are not taking in by trivialization.

Iterators and generators

A solution is to use class libraries and frameworks that allow non-trivial examples to be developed without much knowledge of syntax. Iteration, for example, can be introduced through *iterators* and *generators*—which are packaged as part of a library of container classes, written as templates. The model of iterators used in the DAA course is similar to that of CLU [19]; the same concept essentially is called a generator in Alphard [21]. In the DAA course, the term generator is used for a LISP-like mapping function which applies a given function to each element of a data structure, without having to write a loop.

An iterator produces one value at a time out of a container, and therefore requires a loop to use it in a program. For this reason, a generator is introduced first, which allows a given action to be applied to every element in a container without writing a loop. In this way, it is possible to introduce a range of object-oriented concepts, including instantiation of templates, inheritance and the difference between classes and objects, well before much language syntax is covered. Also, long before students see how a specific container is implemented, they understand that different containers can be used in the same way, if the abstraction is so designed, so the choice of container becomes an efficiency issue which can be deferred as long as possible.

Here is how a generator is implemented. A container class, which is a template parametrized for contents type `Contents`, implements a generator through a class member function, `generate`, of the following type:

```
void generate (T_Action <Contents> &actor)
```

The generator uses a reference argument of the template class `T_Action`, instantiated for the same contents type as the container. Since a reference is in effect a pointer, it's possible to pass an object of a class derived from `T_Action <Contents>`. For a given contents type, it's possible to define any number of action classes which can be used to create an action object to pass to a generator. For instance, for contents type `int`, an action class could be defined to add every integer presented to it by the generator; another action class could be defined to print every integer presented to it by a generator.

An action class has three member functions, which define how the generator processes a container's contents:

- `first_time`—executed before any data is accessed (even if no data in the container)
- `completion`—executed after all data is accessed (even if no data in the container)
- `do_each`—executed once for each element of the container, each of which is passed in as an argument

Of these, only `do_each` has to be defined (in C++ terminology, it's a pure virtual function); the other two functions default to no action.

The generator is a powerful concept, and requires an in-depth understanding of classes to appreciate fully. However, once learnt, students can use the idea on any data structure. Although other class libraries may implement generators differently—or not at all—one fundamental idea is important: it's possible to have an external interface to a container that looks the same across a variety of different containers.

The case for toy libraries

To make all this work requires reasonably well-developed class libraries, which can be used for non-trivial examples. Commercial-quality libraries are generally too large and complex to fit into a course where the focus is not on learning the libraries themselves, so libraries specific to the DAA coursed have been developed. To keep complexity under control, they are divided into three categories: container class templates, a window-based application framework, and toolboxes for specific purposes (database, graphics, text manipulation and strings). Each toolbox can be used separately for specific examples, or combined for larger examples; by the end of the course, the students have implemented part of an example using all the libraries in one application.

One concept in the new DAA course which is totally new relative to the older courses (first version of DAA and the previous Modula-2 course) is the idea of a *software architecture*—the high-level design of overall structure and flow of control [14]. The main example used is the Smalltalk Model-View-Controller paradigm. It would have been difficult to illustrate the idea of an application architecture without an application framework as a basis for implementing that part of the architecture

which does not vary from application to application. Other architectures which are described briefly include client-server architectures, a software bus and compound document architectures (such as OpenDoc).

Gains over original DAA

Compared with the previous version of the DAA course, the latest version introduces templates early, whereas the previous version didn't use them at all, as they were poorly implemented in most compilers in 1994, when the course was being prepared. The previous version did not use libraries; the new one did. Container classes are introduced early in the new version, versus late in the old version. The original version of DAA treated an introduction to software engineering as an extra section at the end of the course, while the new version used software engineering as a basis for explaining the need for algorithm analysis, and invoked principles such as information hiding and planning for testing from an early stage.

On the whole, the later version of DAA covers considerably more ground than the original, which supports the case that the abstraction-first approach aids learning.

Appendix A contains a table of contents of the original DAA, while Appendix B contains that of the latest version. While the original version had some aspects of promoting abstraction from an early stage, there is a clear difference in the latest version, in using the abstraction idea more comprehensively to order topics. The discussion here is intended to convey the essential difference between the two versions of the course; Section 5 contains a comprehensive list of additional material that was covered in the latest course.

4 C++ Traps and Pitfalls

C++ presents some problems in presenting concepts in an abstraction-first order.

Examples covered here are memory management, templates (for parametrized types), exceptions and general complexity of the language.

As a language with programmer-managed memory and pointers, C++ presents problems of aliases, dangling pointers, and potential memory leaks. At an earlier stage than is desirable in terms of the abstraction-first ordering, therefore, it necessary is to cover the relatively advanced topics of deep and shallow copy, including use of copy constructors and overloading the assignment operator.

Deep copy is copying an entire object, including allocation of new memory for copies of dynamic data to which the data structure has pointers. Shallow copy is a bitwise copy, which results in aliases to any data accessed through pointers in the original object.

Templates—used from an early stage—present a number of problems, while exceptions are hard to use consistently without explaining them in full very early.

The remainder of this section presents C++ problems in more detail.

Example of alias problems

Aliases can cause many problems for the unwary; here is a simple example to illustrate some of the problems.

Consider the following class, an object-oriented wrapper around a C-style string:

```
class String
{public:
     String (char new_data [ ]);
     virtual ~String ();
     bool operator == (String &other);
     bool operator < (String &other);
     char *get_chars ();  // pointer to actual data
     int get_length ();
private:
      char *data;
      // size is length+1, to allow for final '\0'
      int length;
};
```

An object of class `String` contains a pointer, which is allocated in the constructor:

```
String::String (char new_data [ ])
{    length = strlen (new_data);
     data = new char [length + 1];
     strcpy (data, new_data);
}
```

It would seem to be obviously correct—without a deep knowledge of semantics of pointers—that the destructor should deallocate the pointer, as follows:

```
String::~String ()
{    delete [ ] data;
}
```

Unfortunately, if a `String` object is copied, an alias results (since the internal pointer is copied bit for bit, with the default shallow copy). If either copy of the object is deallocated, the destructor deallocates the pointer, leaving a dangling pointer in the other copy. Such copies can occur in obvious places, like assignments—but also in less obvious places, like when a value argument (the default parameter-passing mechanism in C++) is passed. It's possible to work around the problem by defining a copy constructor and overloading the assignment operator for the `String` class, to ensure that a deep copy occurs, but these relatively advanced concepts have to be introduced relatively early, to avoid pointer hazards.

Template problems

Another big problem in practice is that templates can produce a cascade of confusing error messages, which require understanding of the implementation, if they are not instantiated correctly. Part of this is a problem with current compiler technology, but it appears that the semantics of templates cannot be checked fully until they are instantiated. Consequently, the compiler cannot easily distinguish between instantiation errors, and errors which are in the template itself but which can only be picked up at instantiation time.

Exceptions

Exception handling in C++ also presents problems. If exceptions are used, they have to be handled consistently. If they are not, unexpected errors can occur through an exception being propagated, and handled in the wrong place. As a result, a design decision has to be made for the course: do not use exceptions at all (except where they can be hidden within an implementation), or introduce them very early, at risk of having to introduce too much detail too soon. The compromise taken in the course was to explain why they were problematic, and to use them only where they could be completely isolated into the implementation of a library. The alternative of introducing exceptions very early (to enforce consistent usage of them) would mean introducing a difficult implementation concept early in the course. By comparison, the CLU model of exceptions [19] converts any exception propagated out of a single routine to a catch-all `failure` exception. In a CLU-like model of exceptions, it would be much easier than in C++ to use exceptions as if they were normal run-time errors in the early part of the course, and introduce explicit exception handling at a later stage.

General problems and summary

A final problem is that some of the detailed rules of the language can be confusing to relative beginners. A frequent occurrence is that students wrap sections of code in unnecessary braces (the C++ equivalent of **begin-end** in Pascal) because they aren't totally sure of the rules for where new variables may or may not be defined.

On the whole, C++ requires more defensive programming and more care in the choice of the order of presentation of concepts than Pascal or Modula-2. It's likely that other object-oriented languages would be easier to use to introduce concepts. For example, Smalltalk-80 [15] and Java both avoid pointer problems by hiding memory allocation and using garbage collection. However, for reasons such as access to reasonably priced compilers with good environments[1], and externally imposed requirements for exposure to C++, many educators are likely to be faced with the need to use C++ as a first object-oriented language. If Java grows beyond its flavour of the month status, it may be a viable alternative. Its lack of parametrized types may be seen by some an obstacle to developing concepts like container classes, but as is often the case with switching to a different language, such obstacles can be overcome by using language features not available in the original language (in this case, interfaces).

[1] For DAA, Metrowerks CodeWarrior on Power Macs is used; since the course was designed, Java appeared and became widely accessible—including free compilers. CodeWarrior has since been ported to other platforms, and now includes Java (as well as Pascal, C and C++).

5 Evaluation: Abstraction-First versus Initial DAA

Compared with the first attempt at the DAA course, in which the abstraction-first idea was not pursued as strongly, a number of new concepts, some of which had previously been covered in more advanced courses, could be introduced.

The new concepts in the abstraction-first version of the DAA course include:

- templates
- iterators
- generators
- deep and shallow copy (including reference counts)
- exceptions (if mainly from the perspective of problems with the C++ model)
- more challenging algorithms and data structures
- use of an application framework
- software architectures

Despite covering a number of new sophisticated topics in a course of the same duration, students' results did not differ significantly between the two years. The average result for the first DAA course was 63%; the average final result for the new course was 62%; in both cases, the standard deviation was 12%.

The students' results in the two C++ courses are in line with other results in our undergraduate courses. However, the group which took the new DAA course performed below the norm in their other courses, so it is unlikely that the fact that they were able to cope with more concepts is attributable to this being an unusually good group of students.

This evaluation supports the claim that the abstraction-first approach is not confusing and difficult for students. A more comprehensive evaluation is needed to make stronger claims, such as that the abstraction-first approach is likely to result improved ability to practice reuse in the workplace.

6 Conclusions

The new DAA course covered a lot of ground, and the abstraction-first ordering made it possible to move a significant number of new advanced concepts to that course, freeing up space in other courses for new concepts. In addition, our graduates have a broader view of Software Engineering than would have been the case had we seen the third-year Software Engineering course as a module on its own, with no linkages to other courses.

Other object-oriented languages may well be better than C++ for introducing concepts. Languages like Smalltalk or Java, for example, which have implicit

memory management and garbage collection, would remove many of the pitfalls of C++.

Whichever language is used, the fundamental principles remain the same: concepts should be introduced in an order which introduces and reinforces reuse from the start. Development from scratch should be seen as a more advanced skill, learnt only after reuse and design are understood—to promote design for reuse. The way to develop course materials which support such a strategy is to start from comprehensive (if toy by industrial-strength standards) class libraries. The classes should contain sufficient functionality and application-building tools to allow introduction of a broad range of object-oriented concepts, before too much language syntax needs to be introduced.

To illustrate how these principles translate to text book design, a table of contents from the latest C++ course is included as Appendix B; for comparison, the previous C++ course's table of contents is included as Appendix A.

How well this strategy will translate to work place skills will take time to evaluate; the widely acknowledged problem in industry of converting programmers to reuse suggests that a change in educational strategy is important to consider.

References

1. 'A Summary of the ACM/IEEE-CS Joint Curriculum Task Force Report: Computing Curricula 1991', *Comm. ACM*, **34**(6):68–84 June 1991.
2. R M Adler. 'Emerging Standards for Component Software', *Computer*, **28**(3):68–76 March 1995.
3. K Auer. 'Smalltalk Training: As Innovative as the Environment', *Comm. ACM*, **38**(10):115–117 October 1995.
4. W Berg, M Cline and M Girou. 'Lessons Learned from the OS/400 OO Project', *Comm. ACM*, **38**(10):54–64 October 1995.
5. G Booch. *Object-Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1991.
6. T A Budd. *Classic Data Structures in C++*, Addison-Wesley, Reading, MA, 1994.
7. J O Coplien. *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.
8. B J Cox. *Object-Oriented Programming: An Evolutionary Approach* (second edition), Addison-Wesley, Reading, MA, 1991.
9. H M Deitel and P J Deitel. *C++ How to Program*, Prentice Hall, Englewood Cliffs, NJ, 1994.
10. M E Fayad and W-T Tsai. 'Object-Oriented Experiences', *Comm. ACM*, **38**(10):51–53 October 1995.
11. W Ford and W Topp. *Data Structures with C++*, Prentice Hall, Englewood Cliffs, NJ, 1996.
12. W B Frakes and C J Fox. 'Sixteen Questions About Software Reuse', *Comm. ACM*, **38**(6):75–87,112 June 1995.
13. S J Garland, *Introduction to Computer Science with Applications in Pascal*, Addison-Wesley, Reading, MA, 1986.
14. D Garlan and D E Perry. 'Introduction to the Special Issue on Software Architecture', *IEEE Transactions on Software Engineering* **21**(4):269–274 April 1995.
15. A Goldberg and D Robson. *Smalltalk–80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
16. M R Headington and D D Riley. *Data Abstraction and Structures Using C++*, DC Heath, Lexington, MA, 1994.
17. E B Koffman. *Pascal* (fourth edition), Addison-Wesley, Reading, MA, 1992.
18. S B Lippman. *C++ Primer* (second edition), Addison-Wesley, Reading, MA, 1991.
19. B Liskov, R Atkinson, T Bloom, E Moss, JC Schaffert, R Scheifler and A Snyder. *CLU Reference Manual*, Springer, Berlin, 1981.
20. R Sedgewick. *Algorithms in C++*, Addison-Wesley, Reading, MA, 1992.
21. M Shaw (editor). *ALPHARD: Form & Content*, Springer, New York, 1991.
22. B Stroustrup. *The C++ Programming Language* (second edition), Addison-Wesley, Reading, MA, 1991.
23. A M Tenenbaum and M J Augenstein, *Data Structures Using Pascal* (second edition), Prentice-Hall, Englewood Cliffs, NJ, 1986.
24. P S Wang. *C++ with Object-Oriented Programming*, PWS, Boston, MA, 1994.
25. R S Wiener and LJ Pinson. *The C++ Workbook*, Addison-Wesley, Reading, MA, 1990.

Acknowledgments: Much of the work on the Modula-2 course was done by Scott Hazelhurst, which saved me time and effort in subsequent years. I reused some of his material in the C++ DAA course. Several generations of students contributed to my understanding of how to present concepts, leading to the proposals in this paper. I would like to thank Apple Computer and Metrowerks for making this course possible on a tight budget.

Appendix A: Original C++ Data Abstraction and Algorithms Course

Chapter 1 Introduction

Chapter 2 Abstraction
- 2.1 user example
- 2.2 programmer example

Chapter 3 Extending an Abstraction
- 3.1 classes
- 3.2 simple class example
- 3.3 bigger class example

Chapter 4 Implementing Abstractions
- 4.1 using classes
- 4.2 defining classes
- 4.3 scope and binding: introduction
- 4.4 the life of an object
- 4.5 extending an abstraction
- 4.6 dynamic dispatch
- 4.7 abstraction and efficiency
- 4.8 algorithm analysis introduced
- 4.9 finding a better algorithm
- 4.10 C++ syntax—first look at detail

Chapter 5 Abstract Data Types
- 5.1 general concepts and examples
- 5.2 more specific examples (stack, array, hash table, tree)
- 5.3 more detail of tree
- 5.4 C++ implementation: tree and queue
- 5.5 C++ more C++ detail: parameter passing
- 5.6 C++ yet more C++ detail: member function calls
- 5.7 final C++ detail: arrays and function calls
- 5.8 data structure checklist

Chapter 6 Container Classes and Algorithm Analysis
- 6.1 generalized model
- 6.2 multiple inheritance and mixins
- 6.3 back to algorithm analysis
- 6.3.1 sorts
- 6.3.2 other important operations: search
- 6.4 implementing a general container
- 6.5 extra reading

Chapter 7 Object-Oriented Analysis and Design
- 7.1 introduction to object-oriented analysis and design
- 7.2 structure of requirements document
- 7.3 object-oriented design
- 7.4 overall goals of analysis and design

Appendix B: Final C++ Data Abstraction and Algorithms Course

For the latest version of the DAA course, see *<http://www.cs.wits.ac.za/~philip/books/daa.html>*.

Chapter 1 Introduction
- 1.1 Introduction
- 1.2 Part 1—Using Abstraction
- 1.3 Part 2—Implementing Abstraction
- 1.4 Part 3—Design and Generalization

Part 1 Using Abstraction

Chapter 2 Abstraction
- 2.1 Introduction
- 2.2 Different Abstractions of the Same Thing: A Bus
- 2.2 A Virtual Machine: The Macintosh User interface
- 2.3 Summary

Chapter 3 Abstraction and Programming
- 3.1 Introduction
- 3.2 Abstract Data Types
- 3.3 Examples of Libraries
- 3.4 A Framework
- 3.5 Putting It All Together: A Complete Application

Chapter 4 First Look at C++
- 4.1 Introduction
- 4.2 What Goes Where
- 4.3 C++ Class Definitions
  - 4.3.1 Syntax for Classes
  - 4.3.2 Inheritance
  - 4.3.3 Scope and Binding
- 4.4 More C++ Syntax
- 4.5 Putting it All Together: Iterators and Generators Revisited
- 4.6 The Big Picture: Managing the Software Process

Part 2 Implementing Abstractions

Chapter 5 Classes and Objects
- 5.1 Introduction
- 5.2 Dynamic Allocation: More on Constructors and Destructors
- 5.3 Inheritance and Dynamic Dispatch
- 5.4 Arguments: More Scope
- 5.5 Operators and Overloading—And the Shallow Copy Hazard
- 5.5 Static Members: Still More Scope
- 5.5 Generalization: Templates versus Abstract Classes
- 5.6 Putting It All Together: Libraries and Frameworks Again

Chapter 6 Object-Oriented Design
- 6.1 Introduction
- 6.2 E-Booch Diagrams Revisited
- 6.3 A Design Example
  - D2.2.4.1.1 Design Decisions
  - D2.2.4.1.3 Major Functionality
  - D2.2.4.2 Module Model: StudentID (from complete design)
  - D2.2.4.2.3 Major Functionality
  - D3 Detailed Design of Each Module
  - D3.2.4.2 Module Model: StudentID (from complete design)
  - D3.2.4.2.1 Member Functions of StudentID
- 6.4 Implementation Example
- 6.5 More Difficult Design Decisions
- 6.6 Conclusion

Part 3 Design and Generalization

Chapter 7 Complexity Analysis Introduced
- 7.1 Introduction
- 7.2 Design Choices
- 7.3 General Principles
- 7.4 Simple Examples without Recursion

7.5 Simple Example with Recursion
7.6 Data Structures
7.7 Bigger Example: Naïve Convex Hull
7.8 Putting It All Together: Relation to Detailed Design

Chapter 8 Libraries and Frameworks
8.1 Introduction
8.2 Design for Generality
8.2.1 Software Architecture
8.2.2 Decide on the Abstraction
8.2.3 Example
8.3 Designing a Template
8.4 Shallow Copy Made Safe
8.5 Handling Errors: Exceptions
8.6 Putting It All Together: A New Container Class

Chapter 9 More Advanced Complexity
9.1 Introduction
9.2 Average Case of Another Simple Algorithm: Bubble Sort
9.3 Recurrence Relations and Closed Form
9.4 Generating Data to Force Best or Worst Case
9.5 Example of Optimality
9.6 More Advanced Data Structures—Hash Tables, Balanced Trees
9.7 Convex Hull Revisited: Graham Scan
9.8 More Interesting Algorithms—Encryption
9.8.1 Cæsar's Cipher
9.8.2 Monoalphabetic Substitution
9.8.3 Polyalphabetic Substitution
9.8.4 Transpositions
9.8.5 Data Encryption Standard (DES)
9.8.6 Public Key Encryption
9.8.7 The RSA Algorithm
9.8 Conclusion

Received 6/97; Acccepted 29/7/97; Final copy 15/8/97.