A Strategy for Teaching Programming from a Distance

Karen Renaud, John Barrow & Petra le Roux
University of South Africa
{renaukv,barroje,lrouxp}@unisa.ac.za

June 21, 2001

Abstract

The department of Computer Science and Information Systems at the University of South Africa (Unisa) has unique problems and challenges in teaching programming from a distance-education perspective. This paper will discuss the process whereby we recently decided on a new strategy. We discuss the constraints, problems and criteria which shaped our reasoning process. We will expound upon our goals and give an outline of our eventual solution.

1 Introduction

Programming is an integral and essential part of the computer science curriculum [17]. It is impossible to become complacent and in this field there is no such thing as an unchangeable 5-year plan. Our department recently decided to review our strategy for teaching programming to our BSc students. This was triggered by deadlines with respect to the University calendar, problems with respect to the availability of one of our current languages and most importantly a need to keep up with the evolving technology in our field. Regular course revision forms an integral part of an ongoing quality control process in our department and the above-mentioned triggers served to indicate an area of focus.

The rest of the paper will take the reader through our decision making process and the rationale behind our proposed new teaching strategy. Section 2 will give some background about our current teaching strategy and explain why we felt it necessary to revise our course structure. Section 3 will discuss the decision process in changing the course content. This section also gives an insight into some of the constraints and criteria we were subject to, as well as an idea of our goals and problems faced in achieving them. Section 4 gives an outline of our our proposed solution, and Section 5 concludes.

2 Current Teaching Strategy

The current teaching strategy starts off with a bridging module, currently using Pascal, which teaches students basic structured programming. Students who have done programming at Secondary School level are exempted from this module.

We then teach object-oriented programming (using C++) and visual programming (using Delphi) at the first year level. The students continue to the second year level to learn about data structures and algorithms and

more object-orientation – still using C++. Historically, the third year introduces some advanced programming concepts, also using C++ but with the expansion of C++ at the second year level, this is due for revision.

The current situation is shown in Figure 1. The introduction of a visual programming module using Delphi as an elective at the first year level was a controversial step but the enrolment for this module has been staggering (1200 students), exceeding all expectations, indicating a definite demand for this programming paradigm. The use of a sophisticated visual environment such as Delphi for teaching programming in the first year, especially in a distance-teaching environment, has some sound pedagogical justifications, similar to those propounded by Rasala [22] for toolkits. We find that at the end of three years our students are not merely adequate, but good,

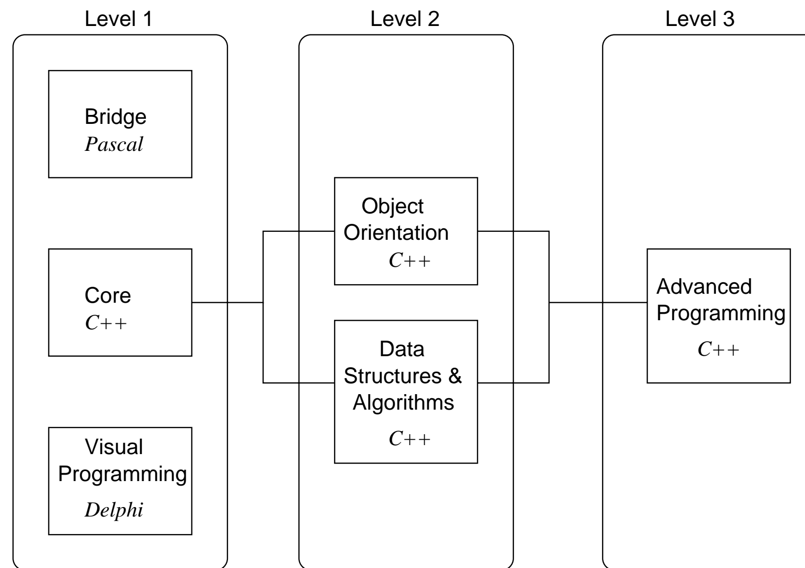

Figure 1: Present Course Structure

programmers and comparable to programmers produced by the other Universities in South Africa. However, there were some problems with our model:

1. There are problems obtaining and using Pascal, especially with the newer operating systems.

2. Students do up to three different programming modules in their first year, each using a different language (Pascal, C++, Delphi/Object Pascal) and some lecturers felt that this was not ideal.

3. Students want to learn Java.

4. Our students did not progress to distributed programming concepts, primarily because we changed programming languages at the 3rd year level. This was felt to be a deficiency in our current course structure which had to be addressed.

5. Our current course structure teaches programming in isolation, independently of other modules [25] which would benefit from a practical component, such as databases [18], operating systems [7], networks [10], human-computer interaction (HCI) [12], artificial intelligence [13] and distributed computing [24].

The following section will discuss the process of arriving at an alternative course structure.

3 Process

Section 1 aluded to various events which triggered a process of change in our department. One of the triggers was the submission date for the University Calendar entries. To meet this deadline, a meeting was hurriedly convened between lecturers who were interested in the programming syllabus in the department. It was evident that everyone in the meeting had a position to defend and there was a clear 'win-lose' mindset. With the pressure of needing to make a hurried decision, the long-standing members of the department felt impatient with the lack of appreciation of the 'important' issues among the newer members. These included the following factors:

- the need to introduce change far more deliberately and over a longer time span in a distance education environment than in a residential institution,
- Unisa's 'open access' policy and the resulting bridging and foundational support needed, and
- the fact that a five year plan had recently been adopted after much heated debate within the department and that we were only on year two of this plan.

The newer members of the department felt that the others were merely bulldozing through their prejudices and were completely closed to any new perspective. We found that it was impossible to reach consensus. The option facing the management committee was either to enforce an autocratic decision, so entrenching divisions and resentment within the department or to negotiate an extension of the calendar deadline and to set up a process aiming for consensus.

The latter option was adopted and a series of weekly meetings scheduled. The whole matter had also become the centre of considerable informal discussion between lecturers in the department and so it became important for the continued health of departmental relationships for the matter to be resolved.

The following meeting removed the extreme goal orientation that characterised the first meeting. The participants spent time establishing the history of programming tuition within the department and elaborating on existing rationale. The first important milestone was that we established the importance of focusing on *what* has to be taught rather than on the programming language used to convey the concepts to the students. Thus we decided to develop criteria, conceptual basis and constraints without mentioning the C-word (C++), the D-word (Delphi) or the J-word (Java). Everyone had the opportunity to express his or her opinion without time pressure. The following sections will report on the criteria and constraints identified during this meeting.

3.1 Factors to be Considered

1. *Number of programming modules* — this is an important issue — as indicated by the furore over the recent ACM proposal with respect to the core computer science curriculum which proposed only 5 hours programming tuition. It was felt that since most of our graduates would start their career programming it was important that we exposed them to a good deal of programming during their undergraduate degree. This issue can be considered from two perspectives:

    (a) Per year

    (b) In total

2. *Number of programming languages in undergraduate course.* This question provoked much debate. The programming language issue includes the following points:

   (a) the choice of a first programming language [17],

   (b) whether a language can be used to teach more than one paradigm. In 1993 Mancordis *et al.* [15] cited three main problems in teaching computer science:

      i. a lack of tool integration,

      ii. a lack of tools for programming activities other than coding, and

      iii. the proliferation of programming languages and tools.

      The first two problems are no longer problems in the 21st century but the latter still causes headaches. It is difficult to know whether a language is going to become a legend (like COBOL), or whether it is going to evaporate quickly (like PL/1). Departments need to maintain a balance between enriching the curriculum by exposing students to a variety of languages, and confusing them by giving them access to too many.

   (c) how many languages the students should be exposed to [13].

3. *Support for other modules.* It was felt that other modules should have the freedom to build on programming modules should they require practical assignments to be completed.

4. *Paradigms to be taught* [17] —

   (a) Visual [11],

   (b) Object Orientation [20],

   (c) Structured Programming [8],

   (d) Client/Server and the Internet [3, 27, 24],

   (e) Design Patterns [2, 19],

   (f) Parallelism [1, 14] and Concurrency [7],

   (g) Logic [21], and

   (h) Functional [26].

5. *Relationship between concepts and application.* It seemed that our current course structure attempted to teach both concepts and application within the *programming* modules and that this limited the practical application of the concepts we were trying to teach.

6. *Foundation support, level of incoming students and appropriate depth.* Unisa does not require students to have done Computer Studies (programming) at school before enrolling for our programming modules. This means that we cannot expect any level of computer literacy in our students. This requires the teaching to start at a very low level since foundation support is essential is maintaining exit standards. Since we start at a lower level than some other Universities it is a concern that we do indeed bring students up to the required standard.

7. *Industry forces.* This is a controversial point, and one which will surely never be resolved. Universities do have to take these forces and demands into account, but not at cost to the excellence of our teaching. We do not want to become merely a service industry which delivers the skills that industry requires at any time, but rather strive to produce graduates who have learnt how to think and who can develop the skills required by industry. There needs to be a balance between assimilation of long-term concepts and the learning of short-term skills. A long-term knowledge base ensures that students are adaptable to current organisational needs.

3.2 Constraints

1. *Distance education model* — Changing the structure of a course in an institution such as Unisa is not comparable to the same process in a residential University [6]. Students can take as long as 10 years to complete a degree so any changes to programming modules take some time to filter through to students at each level of the course. Our students are often working full time and studying at home which means that they do not have the benefit of learning vicariously from other students. Although Unisa does provide laboratories with tutors many students who are in full-time employment will study at home. This means that many students will not have a tutor to help them out and they have to wait till the following day to contact a lecturer to assist them. As a result our students need nurturing and special assistance to alleviate the isolation and frustration they often experience in attempting to acquire programming skills in this type of environment.

2. *Resources* — Unisa students often come from impoverished backgrounds and cannot afford the latest technology. While Unisa does have laboratories at all the major centres we are not able to offer facilities to all our students throughout South Africa and so we have to take this into account when deciding on a technology. For example, we cannot assume that students have Internet access even though many will have it at work or be able to make use of an Internet café. Students at Unisa also, unlike students at residential Universities, have to provide for their own computing needs. Thus ease of use and robustness are vital because they have to install the software themselves and learn how to use it without any assistance.

3. *Rate of technological change* — One of the major problems in our field is that one cannot become complacent once one has identified a good programming language and developed a course around it. Operating systems change and compilers become unavailable or do not run on new operating systems and upset the best attempts of lecturers to provide students with a stable enabling technology.

3.3 Goals

Our final goals are to teach students [5, 23]:

1. to master core programming skills,

2. the ability to reason logically and solve problems,

3. to implement solutions to those problems,

4. the ability to synthesise ideas,

5. to be professional,
6. the ability to communicate ideas to others, and
7. the desire and ability to develop and educate themselves further.

We would like to teach students the skills necessary to enable them to adapt to the needs of a particular organisation. However, computer science is not only about programming, and students need to see programming within the broader perspective of computer science [9, 23, 17].

4 Proposed Solution

As a result of the real progress achieved at the second meeting and the realisation that a 'win-win' situation was possible, people softened their stances, were prepared to listen to others' opinions and acknowledged both the validity of others' positions and weaknesses in their own. Through attempting to accommodate as many perspectives as possible, the department was led to rethink the entire syllabus, not just the programming curriculum, and this has resulted in a considerably more flexible and adaptive structure. We believe this to be the case because:

1. programming skills are taught up to the second level, after which the students are expected to apply these skills in a practical manner in a variety of different application areas. This enables the student to hone their skills and apply them in an abstract manner.
2. students are free to make use of the language most suitable for the practical project set by each module and are not restricted to the use of any particular third-level language.
3. moving the teaching of programming skills to the second level makes space for students to take a double major at the third-year level.

The proposed solution is shown in Figure 2. We decided to retain C++ as our primary language. C++ is still widely used [16] although many CS departments are moving to Java. It was felt that a change to Java would not be a good choice for distance-education students although everyone in the department agreed that Java *should* be introduced in the curriculum. C++ is also unsurpassed in teaching students about pointers and memory leaks, problems which simply do not exist in Java.

We also retained both the bridging module and the core programming module at the first year level. This is in line with existing knowledge in pedagogical circles which expouse the teaching of building blocks first (basic programming constructs) before embarking on abstract and subtle concepts (object-orientation) [4].

It is clear that one cannot teach every possible paradigm within the programming language curriculum. Our model allows us to use the skills students have assimilated in the first and second year to teach specific paradigms within the particular third year module which applies such paradigms. For example, the inclusion of an event-driven programming module, using Java, in the second year of our curriculum has satisfied the demands of students to learn a language they perceive to be in demand in the marketplace, and also provides a foundation upon which third-year modules, such as HCI and Distributed Systems, can build.

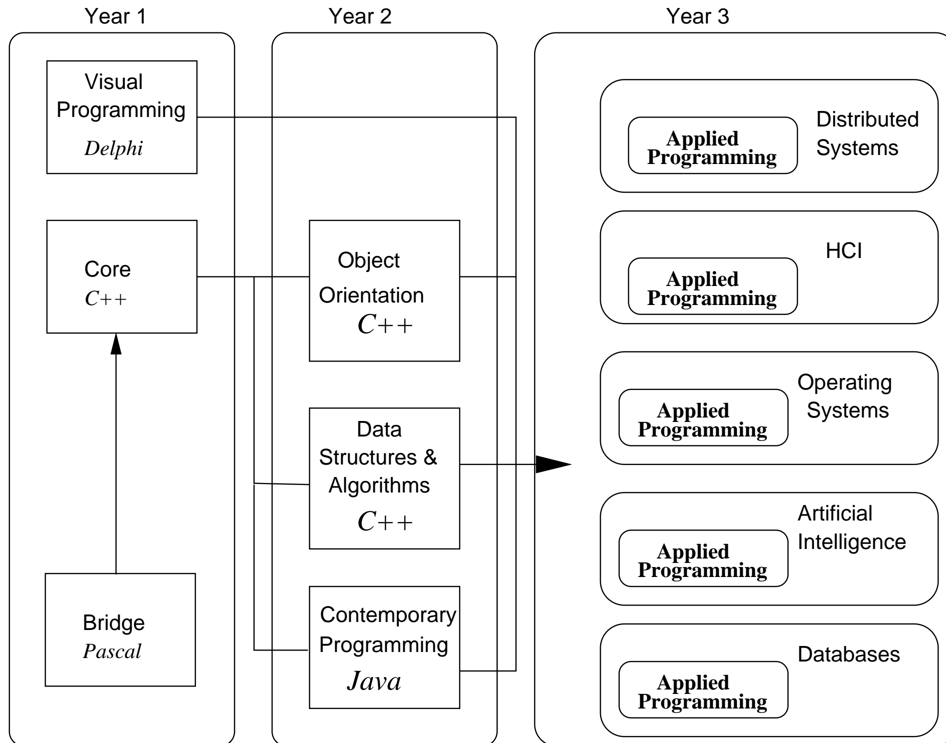

Figure 2: Proposed Course Structure

5 Conclusion

Our experience has shown that it is possible to achieve consensus in a department whose members hold widely divergent views on teaching programming at an undergraduate level. We feel that we have found a way of achieving such consensus without marginalising any members of the department and in such a way that people feel positive about the eventual result.

Finally, the new teaching strategy allows us to teach concepts throughout our courses rather than isolating them within the programming curriculum. We will be monitoring the progress of students through this new curriculum to ascertain whether they benefit as much as we anticipate. We also hope to confirm our expectation that this new model will reduce third-level drop-out rates.

References

[1] M. Allen, B. Wilkinson, and J. Alley. Parallel Programming for the Millennium: Integration Throughout the Undergraduate Curriculum. In *Conference Proceedings Second Forum on Parallel Computing.*, DoubleTree Islander Hotel, Newport, RI, June 22 1997. www.cs.dartmouth.edu/FPCC.

[2] O. Astrachan, G. Mitchener, G. Berry, and L. Cox. Design Patterns: An essential component of the CS curricula. In *Proceedings of the 29th SIGCSE technical symposium on Computer Science education*, pages 153–160, Altanta, GA USA, February 26 – March 1 1998. ACM.

[3] C. M. Boroni, F. W. Goosey, M. T. Grinder, and R. J. Ross. A paradigm shift! The Internet, the Web, browsers, Java and the future of computer science education. In *Proceedings of the 29th SIGCSE technical symposium on Computer*

Science education, pages 145–152, Altanta, GA USA, February 26 – March 1 1998. ACM.

[4] D. Buck and D. J. Stucki. Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 75–79, Austin, TX USA, March 7–12 2000. ACM.

[5] R. D. Cupper. Computer science: a proposed alternative track – applied computing. In *Proceedings of the 29th SIGCSE technical symposium on Computer Science education*, pages 25–29, Altanta, GA USA, February 26 – March 1 1998. ACM.

[6] W. Doube. Distance Teaching Workloads. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 347–351, Austin, TX USA, March 7–12 2000. ACM.

[7] M. B. Feldman and B. D. Bachus. Concurrent programming CAN be introduced into the lower-level undergraduate curriculum. In *Proceedings of the conference on Integrating technology into Computer Science education*, pages 77–79, Uppsala, Sweden, June 1–5 1997. ACM.

[8] T. Flaherty. A simple technique to motivate structure programming. In *Proceedings of the 19th SIGCSE technical symposium on Computer Science education*, pages 153–155, Atlanta, GA USA, February 25-26 1988. ACM.

[9] P. Gabbert and K. Treu. Experiments with the Use of Popular Press in the Computer Science Curriculum. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 124–128, Austin, TX USA, March 7–12 2000. ACM.

[10] J. M. González-Barahona, J. Centeno-González, P. de las Heras-Quirós, F. J. Ballesteros-Cámera, and L. López-Fernández. Teaching network programming with Ada and Lower_Layer. In *Proceedings of the conference on TRI-ADA '97*, pages 105–110, St Louis, MO USA, November 9-13 1997. ACM.

[11] R. Jiménez-Peris, S. Khuri, and M. Patiño-Martínez. Adding breadth to CS1 and CS2 courses through visual and interactive programming projects. In *Proceedings of the 30th SIGCSE technical symposium in Computer Science education*, pages 252–256, New Orleans, LA USA, March 24–28 1999. ACM.

[12] J. Kay and B. Kimmerfeld. A problem-based interfaces design and programming course. In *Proceedings of the 29th SIGCSE technical symposium on Computer Science education*, pages 194–197, Altanta, GA USA, February 26 – March 1 1998. ACM.

[13] D. Kumar. Curriculum Descant: How much programming? What Kind? *Intelligence*, 11(4):15–16, 2000.

[14] B. L. Kurz, C. Kim, and J. Alsabbagh. Parallel Computing in the undergraduate curriculum. In *Proceedings of the 29th SIGCSE technical symposium on Computer Science education*, pages 212–216, Altanta, GA USA, February 26 – March 1 1998. ACM.

[15] S. Mancordis, R. C. Holt, and D. A. Penny. A "curriculum-cycle" environment for teaching programming. In *Proceedings of the Twenty-fifth SIGCSE symposium on Computer Science education*, pages 15–19, Indianapolis, IN USA, February 18–19 1993. ACM.

[16] R. McCauley and B. Manaris. Computer Science degree programs: what do they look like? A report on the annual survey of accredited programs. In *Proceedings of the 29th SIGCSE technical symposium on Computer Science education*, pages 15–19, Altanta, GA USA, February 26 – March 1 1998. ACM.

[17] D. McCracken. Programming languages in the computer science curriculum. In *Proceedings of the twenty third technical symposium on Computer Science education*, pages 1–4, Kansas City, MO USA, March 5–6 1992. ACM.

[18] M. Merzbacher. Teaching Database Management Systems with Java. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 31–35, Austin, TX USA, March 7–12 2000. ACM.

[19] V. K. Proulx. Programming Patterns and Design Patterns in the Introductory Computer Science Course. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 80–84, Austin, TX USA, March 7–12 2000. ACM.

[20] J. R. Pugh, W. R. LaLonde, and D. A. Thomas. Introducing object-oriented programming into the computer science curriculum. In *Proceedings of the 18th SIGCSE technical symposium on Computer Science education*, pages 98–102, St Louis, MO USA, February 19–20 1987. ACM.

[21] D. Rayside and G. T. Campbell. Aristotle and object-oriented programming: why modern students need traditional logic. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 237–244, Austin, TX USA, March 7–12 2000. ACM.

[22] R. Risala. Toolkits in First Year Computer Science: A Pedagogical Imperative. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 185–191, Austin, TX USA, March 7–12 2000. ACM.

[23] I. Sanders and C. Mueller. A Fundamentals-based Curriculum for First Year Computer Science. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 227–231, Austin, TX USA, March 7–12 2000. ACM.

[24] D. L. Spooner. A Bachelor of Science in information technology: an interdisciplinary approach. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 285–289, Austin, TX USA, March 7–12 2000. ACM.

[25] B. Toll. The distributed course – a curriculum design paradigm. In *Proceedings of the 29th SIGCSE technical symposium on Computer Science education*, pages 20–24, Altanta, GA USA, February 26 – March 1 1998. ACM.

[26] S. Vandenberg and M. Wollowski. Introducing Computer Science using a breadth-first approach and functional programming. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science education*, pages 180–184, Austin, TX USA, March 7–12 2000. ACM.

[27] A. Yang and Y. Bacher. Using Java the socket interface in teaching client/server programming. In *Proceedings of the 4th SIGCSE/SIGCUE on Innovation and technology in Computer Science education*, page 206, Krakow, Poland, June 27–30 1999. ACM.