
Teaching Programming from a Distance: Problems and a Proposed Solution

Karen Renaud, John Barrow, and Petra le Roux

Department of Computer Science and Information Systems
University of South Africa
P O Box 392
Pretoria 0003 South Africa
<{renaukv,barroje,lrouxp}@unisa.ac.za>

Abstract

Teaching programming is never a simple task. It is a dynamic process and the curriculum often evolves from one year to the next. Teaching programming at a distance-education institution is especially challenging. This paper reports on the process of curriculum planning at the computer science department of a distance-education institution. We address generic issues related to teaching programming and specific problems encountered when teaching at a distance. The paper outlines and motivates our proposed strategy, which encompasses three years of undergraduate teaching.

1. Introduction

Programming is an integral and essential part of the computer science (CS) curriculum [5]. It is impossible to become complacent and in this field; there is no such thing as an unchangeable 5-year plan. Our University (Unisa) is a distance-education institution and teaching programming in this way poses unique difficulties. Whereas many Computer Science departments are dismayed at the difficulties their students experience with learning how to program, one can imagine the difficulties experienced by students trying to master these skills without any support in the form of formal lectures, tutorials or interaction with other students.

Our department recently decided to review our strategy for teaching programming to our CS students. This was triggered by course revision 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 article will take the reader through our decision-making process and the rationale behind our proposed new teaching strategy.

2. Current Teaching Strategy

The current teaching strategy starts off with a bridging module, currently using GNU Pascal, which teaches students basic structured programming. We exempt students who have done programming at secondary school level from this module. Unfortunately, there are compatibility issues between GNU Pascal and some of the newer versions of Windows.

We 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.

Figure 1 shows the current situation. 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 [6] for toolkits.

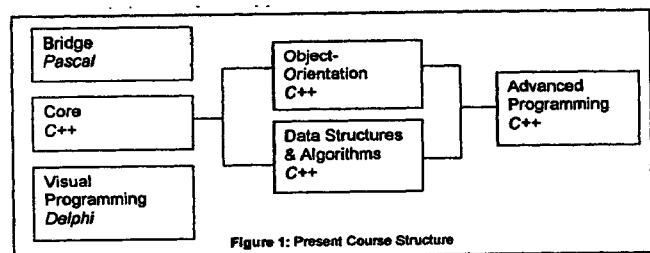


Figure 1: Present Course Structure

We find that at the end of three years, our current students are not merely adequate, they are good programmers and comparable to programmers produced by the other universities in South Africa. However, we have experienced some problems with our existing 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 many lecturers

had misgivings about this.

3. Students want to learn Java.
4. Our students did not progress to learning about distributed programming concepts, primarily because we previously changed programming languages at the third-year level. We felt this to be a deficiency in our current course structure that had to be addressed.
5. Our current course structure teaches programming in isolation, independently of other modules [8] – that would well benefit from a practical component, such as databases, operating systems, networks, human-computer interaction (HCI), artificial intelligence and distributed computing.

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

3. Deciding on a New Course Structure

We alluded to various events that 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, we convened a meeting 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:

1. The need to introduce change far more deliberately and over a longer time span in a distance-education environment than in a residential institution;
2. Unisa's 'open access' policy. That is, the University of South Africa does not require students to write entrance exams – anyone with a school matriculation certificate with the required subjects is accepted for our degree programs. Therefore, it needs the resulting bridging and foundational support; and
3. 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 departmental 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.

We adopted the latter option and scheduled a series of weekly meetings. 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 orien-

tation 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.

4. Factors Under Consideration

1. *Number of programming modules* - this is an important issue - as indicated by the furor over the recent ACM Computing Curricula 2001 draft proposal which proposed only 5 hours programming tuition but which has subsequently been revised upwards [9]. 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. Since our degree programme allows the option of a 'dual major' in both computer science and information systems, we can include more programming modules than is normally the case. One can consider this issue from two perspectives:

- a. Number of programming modules per year.
- b. Number of programming modules in total.

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

- a. The choice of a first programming language [5].
- b. Whether a language can be used to teach more than one paradigm. In 1993 Mancordis *et al.* [4] 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 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. The number of languages students should be exposed to.

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* [5] — These include: Visual, Object Orientation, Structured Programming, Client/Server and the Internet, Design Patterns, Parallelism and Concurrency, Logic, and Functional.

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 us to start teaching at a very low level since foundation support is essential in 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 that 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.

5. Constraints

5.1 Distance education model

Changing the structure of a course in a distance-education institution such as Unisa is not comparable to the same process in a residential University [3]. 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 until 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.

5.2 Laboratory Facilities

Unisa students often come from impoverished backgrounds and cannot afford the latest technology. While Unisa does have laboratories at 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.

5.3 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.

6. Goals

Our final goals are to teach students the following [2,7]:

1. Master core programming skills,
2. Develop the ability to reason logically and solve problems,
3. Implement solutions to those problems,
4. Develop the ability to synthesise ideas,
5. Be professional and ethical,
6. Develop the ability to communicate ideas to others, and
7. Create 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.

7. 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 that of their programming curriculum. 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 increase their

level of specialisation at the third-year level.

Figure 2 shows the proposed solution.

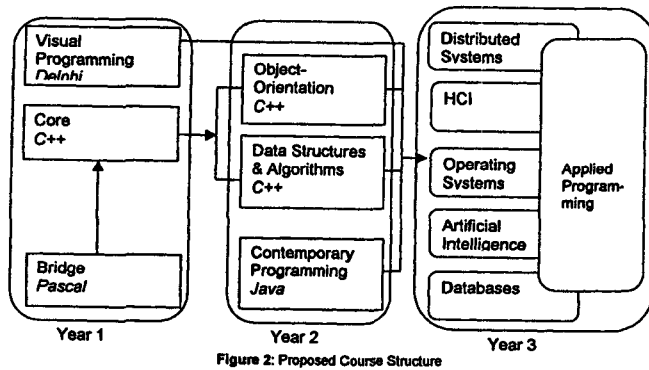


Figure 2: Proposed Course Structure

We decided to retain C++ as our primary language. C++ is still widely used 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 that 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 espouses the teaching of building blocks first (basic programming constructs) before embarking on abstract and subtle concepts (object-orientation) [1].

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 a contemporary 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.

8. 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 depart-

ment 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 dropout rates.

References

- [1] 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.
- [2] 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, Atlanta, GA USA, February 26 - March 1 1998. ACM.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] R. Rasala. 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.
- [7] 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.
- [8] B. Toll. The distributed course - a curriculum design paradigm. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 20-24, Atlanta, GA USA, February 26 - March 1 1998. ACM.
- [9] Computing Curricula 2001: En Route to the Steelman Draft. Report presented at ITiCSE2001 Conference, Canterbury, England, June 24, 2001. ACM.