Problem-Based Learning for Foundation Computer Science Courses

Mike Barg, Alan Fekete, Tony Greening, Owen Hollands, Judy Kay, and Jeffrey H. Kingston
Basser Department of Computer Science
The University of Sydney
{ mbarg, fekete, tony, owen, judy, jeff }@cs.usyd.edu.au

Kathryn Crawford
SMITE Research Unit
Faculty of Education
The University of Sydney
crawfork@edfac.usyd.edu.au

Abstract

The foundation courses in Computer Science pose particular challenges for teacher and learner alike. This paper describes some of these challenges and how we have designed Problem-Based Learning (PBL) courses to address them.

We discuss the particular problems we were keen to overcome: the pure technical focus of many courses; the problems of individual learning and the need to establish foundations in a range of the areas which are important for computer science graduates. We then outline our course design, showing how we have created Problem-Based Learning courses.

The paper reports our evaluations of the approach. This has two parts: assessment of a trial, with a three-year longitudinal follow up of the students; reports of student learning improvements after we had become experienced with full implementation of PBL.

We conclude with a summary of our experience over three years of PBL teaching and discuss some of the pragmatic issues of introducing the radical change in teaching, maintaining staff support and continuing refinement of our PBL teaching. We also discuss some of our approaches to the commonly acknowledged challenges of PBL teaching.

Keywords: Problem-Based Learning, large first year classes, life-long learning

1. Introduction

The foundation courses in Computer Science pose particular challenges for the teacher: they develop basic skills and attitudes which are important for effective learning in later courses; they are often large courses with correspondingly large management and administrative loads; teaching staff often find them demanding, and for some staff, they are seen as onerous.

Now taking the learner's perspective, consider the critical role of foundation courses.

They give a large cohort of students their first real taste of the discipline. Negative experiences may discourage students from further study. This is a very serious problem if those negative experiences are not indicative of the discipline as a whole.

To make this concrete, consider the role of group and individual work in typical foundation courses. The courses are typically based upon individual work. By contrast, much of the programming workforce operates in teams, where the ability to work co-operatively and to communicate well are important. Students with aptitude and a preference for team work may give up on Computer Science if their first year experiences convince them that the discipline is individualistic.

Consider another example. A large proportion of Computer Science graduates will find employment which involves broad problem solving skills, rather than purely technically centred activity. If the foundation courses have a narrow technical focus, this may deter students with preference for broad problem solving that is so much in demand among graduates. This has been identified as a particular problem for women [8].

In addition, the majority of students in the first year computing courses are also in their first year at University. These students are particularly fragile. Research on the experiences of Australian first year university students indicates that many of these students suffer loneliness and doubt about their choice of degree.

1.1. Motivation for move from conventional teaching

Factors such as these weighed on us in 1995 as we began the redesign of our first year courses. At that point, we like many others, were still teaching Pascal. For several years, we had been acutely aware of Pascal's shortcomings but found the choice of next language very difficult. Even so, at this point, we felt that a move was urgent and we chose to move to an object-oriented language. In fact, we moved to a clean, elegant object-oriented, introductory programming language, Blue [10] [12] [11].

At the same time, we saw major problems with the overall teaching approach and structure of our foundation courses. Our foundation courses had a conventional format, with six contact hours per week, of three lectures, a tutorial and two-hour workshop. We were concerned that this conventional course failed to develop skills needed in later courses. In particular, we wanted the very first courses to place emphasis on three broad classes of skills:

1. generic skills of independent and reflective learning, problem solving, critical thinking as well as written and spoken communication.

2. At the same time, we wanted to achieve increased learning of the technical aspects of programming and introductory computer science aspects like correctness and time complexity analysis.

3. In the middle ground between the purely technical and the generic, we wanted to provide foundations in software engineering areas of requirements analysis, design, planning and coordination of software development and testing, as well as a user-centred view of interactive-software development.

The conventional courses made it very difficult to nurture this range of skills. They generally

achieved the purely technical aspects quite well: the courses had been refined over many years and were a polished form of conventional first year courses.

This range of skills is represented in most full degree programmes. However, it is typically not part of the foundation units. As we have discussed, this means a missed opportunity to show the importance of this range of aspects from the very beginning. It also makes it more difficult to help students establish good habits from the very first courses.

Consider, for example, the teaching of good documentation. In a typical course, where students only write quite small, solo programs over a short period, rarely more than four weeks, it is difficult to help them appreciate the importance of documentation. Certainly, one can set guidelines and explain the reasons for them. One can then tie assessment to good documentation. But one is then left with largely extrinsic motivation for documentation. It is preferable if students can actually experience the benefits of good documentation and suffer the effects of poor documentation. Having this happen at first year makes it easier for students to form good habits which will carry them right through their studies and professional life. Moreover, it avoids students forming bad habits which need to be unlearnt.

Consider another example. In our conventional course, we made extensive use of automated grading. This made our work more manageable. It also constituted a model of thorough testing: we hoped our students would learn from this. However, we observed that it had a different effect. By senior courses, it meant that students expected programming tasks to have extremely tight specifications and they relied on us to set the standards which defined the success of their work. It was surprisingly difficult to teach senior students to set their own testing agenda and standards.

A deeper problem with the conventional course was due to the diversity in our large student body. We took some account of this by teaching the top students in an Advanced form of the course. Even so, this left the 500 to 900 students taking the course in a semester forced to spend their three lecture hours per week in one of two levels of class. Some of the students have studied computing at school or elsewhere. Some have been programming since they were quite young. Most have no programming experience at all. There is also considerable diversity on many other dimensions. Our students come from several degree programmes: Arts, Economics, Education, Engineering, Science, combined degrees such as Science/Law as well as the Computer Science degree. Many study with us for just a semester or two while others will continue to complete a major, Honours degree and postgraduate study. In a monolithic course, it is extremely difficult to cater to this range.

Certainly, there can be some choice in practical tasks. But lectures have to compromise, aiming at the mythical ‘average’ student. If this is well done, we can hope to pace the presentation to about half the class. Around a quarter will find things too fast and another quarter will be bored.

There are other problems with the lecture format for the type of learning that is normally required in foundation computer science courses. The main learning will invariably happen when the students actually write programs and get them working. Even aspects like time complexity analysis need to be learnt by doing the analysis. Year after year, we have seen this problem reflected in student surveys where workshops and tutorials are consistently rated ahead of lectures. As teachers, we felt frustrated at attempting the impossible task of creating lectures suited to the range of students.

Another problem with the conventional course followed from the structure of the practical

work. Like many such courses, there were small weekly exercises as well as two larger assignments per semester. The free form comments in student surveys made it clear that many students saw the weekly exercises as fair but the larger assignments as totally unfair. This may be due to the structure of other courses, like mathematics, where the learning achieved in weekly tasks is adequate for the course goals. We were very concerned that our students be able to integrate the various elements learnt in each week of the course and use them to solve programming problems.

The move to OOP provided additional impetus for re-examining the way that we taught. An important dimension of the OO paradigm comes from the writing of large programs which involve several programmers. If each student does assignments alone, it is more difficult for them to appreciate this aspect. Certainly, one can provide large programs as a starting framework so that the individual work can have more of the flavour of typical OO projects. But this is less satisfactory than group development of a system.

1.2. The choice of Problem-Based Learning (PBL)

As we refined our understanding of the problems with the conventional teaching framework, we identified the elements of a significantly improved structure. The teaching approach which seemed to offer most promise was Problem-Based Learning. Its literature indicates that it does develop just the range of learning we wanted. See, for example, [1]. PBL is most common in professional degrees, especially those in medical and paramedical areas. Seminal work in moving PBL to more technical disciplines has been in the area of Chemical Engineering [17, 18, 19].

Before we describe what this is, we need to explain what it is not. We frequently find people who claim to use PBL but they appear to use this term in a different way from the large and well-established PBL community. For example, most computing courses involve setting 'problems' which students are required to complete. We will refer to these as exercises because they are small and well-defined. We used them extensively in our old conventional course: there were weekly exercises, each focussed on particular detailed aspects of the course, usually one that had been on centre stage in the recent lectures; there were larger assignments which integrated many aspects of the course but were still quite tightly defined, to the point where we could assess their correctness in our automatic grading system.

PBL involves much broader problems which involve a larger set of problem solving skills. Critically, PBL places problem solving and metacognitive skills at the heart of the curriculum. Class time is devoted to such generic problem solving skills as defining a learning plan, brainstorming to get started on a problem, reflection, articulation of problems and solutions, self-assessment, practice in active listening and other communication skills. These aspects are also assessed and contribute to the grade awarded.

Problem-based learning (PBL) is learning by solving a large, real-world problem. Lectures are replaced by additional tutorial and laboratory time. In our case, staff are only present for one hour in the three-hour laboratory class. This is necessary to keep costs similar to those for a conventional course.

PBL has been defined as follows [4]:

> *The principal idea behind problem-based learning is that the starting point for learning should be a problem, a query or a puzzle that the learner wishes to solve…*

> *Problem-based courses use stimulus material to engage students in considering a problem which, as far as possible, is presented in the same context as they would find it in 'real life;' this often means that it crosses traditional disciplinary boundaries. Information on how to tackle the problem is not given, although resources are available to assist the students to clarify what the 'problem' consists of and how they might deal with it. Students work cooperatively in a group or team with access to a tutor who is often not an expert in the field of the particular problem presented, but someone who can facilitate the learning process.*

This notes the role of authentic problems, self-directed learning, cooperative group work and the role of the teacher as facilitator. Another description, from Biggs [3]

> *Learners are assigned to small problem-solving groups and begin interacting with teachers, peers and clients; they build up a knowledge based on relevant material and learn where to go to seek out more. Students meet with a tutor and discuss the case in relation to the knowledge they have obtained. The knowledge is applied, the case treated. Subsequently, there is a review process to ensure that learners develop self-management and self-monitoring skills.*

emphasises the very important role of PBL in developing life-learning learning skills, through various elements of PBL. Critically, PBL courses actively teach generic problem solving skills. They are more than helpful skills which we might hope students bring to their computer sciences. We allocate teaching time to explicit instruction in various generic skills and we assess them. At the same time, we situate the learning within the context of Computer Science.

Examples of the problems we offer include simulating a road network, maintaining information about Olympic events and athletes, and answering arbitrarily complex database queries. The problems are open-ended, and groups are encouraged to research their subject and develop their own specifications and solutions. A different set of problems is set for the Advanced class: all require research into techniques described in the Computer Science literature. For example, here is one of the problems we have offered in the first semester course:

> We are planning to open the Basser Software Mart. This problem requires assistance in planning the check-outs. Some people think that the best thing to do is have a single queue for customers and the person at the head of the queue goes to the first available check out. Other people think it is better to have an Express queue for two check-outs for customers who have less than 6 items and another queue for the remaining four check-outs for all customers.
>
> Your simulation will allow the planners to explore a range of scenarios. For example, customers arrive at different rates, with a burst of them a little after opening time, another burst during lunch time and yet another near closing time. Also, some days are busier than others. It would also be good to explore other possibilities, like cash-only queues.
>
> In your demonstration of the final project, you will show the simulation of two different ways of managing the queues and show how well each does in terms of things you consider important, like average wait time for service, maximum wait time or the like.

Such problems provide a driving force for developing metacognitive skills: students manage and monitor their own learning, and reflect on how to do this more effectively. The first attack on the problem involves students determining the following:

- Problem statement - current understanding of overall goal(s)
- Current subgoals
- How you will know you have succeeded
- What you already know
- Steps to take, by when
- Use of 3-hr class time
- Use of 6-hr private study time

Finally, problem-based learning involves group learning. This serves as part of the development of skills in communication and co-operative work. It also means that students need to discuss their knowledge and approaches and to justify decisions: this externalisation of understanding and knowledge is an aid to improved learning, especially in that case of metacognitive skills.

When we began design of our course, PBL was appealing. However, we were not able to take its pure form where the students spend their whole degree program learning in this format. We needed to adapt it to our two foundation courses in an environment where most students spend only a quarter of their their time in our PBL course, the other three quarters being in various other courses.

In summary, PBL is characterised by:

- open-ended, authentic, substantial problems which drive the learning;
- explicit teaching and assessment of generic and metacognitive skills;
- collaborative learning in groups.

2. Overview of foundation courses

This section gives the flavour of the two courses by describing the way that activities are spread over the semester, the issues associated with group work, assessment and staff development. We describe the first semester in rather more detail so as to communicate the flavour of our approach.

2.1. Semester 1: Introduction to Programming

The approach of the course is to view programming as building models which are implemented as classes, each instance of which corresponds to a real-world entity. The system will repeatedly analyse an event that corresponds to a real-world change, and it will evolve in response to this event.

The course objectives are formally set as follows. By the end of the semester, each student will be able to:

- define a 'simple' class interface and implement it, making effective use of the Blue class library;
- use code quality and testing strategies, including using good style, writing good pre- and post-conditions and class invariants, running and participating in code reviews;
- reason about and explain the design of a systematic, economical and purposeful testing strategy and evaluate the extent and success of a set of tests;
- read and evaluate a class implementation in terms of modularity, code independence, class interfaces, class relationships, cohesion, coupling, overloading;
- utilise the following generic skills:
    - plan learning by formulating the problems to be solved, establishing the things that need to be learnt, and what is already known, defining strategies for learning new things needed, and monitoring progress;
    - self-assess learning by developing strategies for testing new programming notions and making use of supplied self-assessment tools;
    - use reference manuals and other printed material to find information about Blue;
    - use the library and the Internet to find resources relevant to a problem;
    - demonstrate the ability to write an English report about the design of a 'simple' class and its testing, including the purpose for each test and the basis for selecting that as a meaningful;
    - give a well structured oral presentation about the design and testing of the system they have constructed;
    - work co-operatively, using programming-by-contract and communicating with other group members to ensure they know what each is expected to contribute to the group effort and to assess that contribution.

This set of 'Postconditions' is in the Resource Book [Kay1999] and class activities refer to it so that students are conscious of what we hope they will learn by the end of the semester.

There are three main periods in the first semester;

- weeks 1-4, the startup Problem 1;
- weeks 5-11, main work on Problem 2;
- weeks 12-13, reflective period, report writing and demonstrations for Problem 2.

During the first four weeks of the first semester, enrolment is too volatile to form stable groups for problem solving. Instead, we present the first problem as a dry run for what is to come. The

work in this period is assessed entirely individually. However, students work in groups, each group member doing different parts of the problem. A pass on Problem 1 is required before a student is allowed to join a Problem 2 group.

This period also involves several group activities in each week's tutorial. These involve a combination of generic problem solving skills as well as technical skills. For example, an early activity involves practice in active listening with a partner who is doing problem solving. We use generic materials taken from Woods [19] but set them in the context of working out what code fragments do. We carefully choose code fragments: most students will need to work out the answers with the aid of the printed resources. The activity ends with reflection about problem solving style and active listening skills, using Woods' questionnaires. Students are encouraged to change groups for each activity.

When students start Problem 2 in week 5, they know much of their class well enough to form groups which are reasonably compatible. We encourage students to form groups of people with diverse backgrounds, strengths and interests. At the same time, we offer a range of problems and the group has to agree on the problem they will select.

We have already given an example of a Problem 2 task with the Basser Software Mart. All the tasks require simulation of a complex system. All can be designed so that there is a small core of essential code in the main simulation driver classes. Once this is written and working, various other parts of the simulation can begin as very simple stub classes in a working system. These can be upgraded as group members implement more sophisticated versions. The tutors' task is to help guide students to a design which is safe for the group because the essential core is implemented early and no individual student can prevent the group from producing a working system. (This task is not easy: we have been evolving strategies for assisting tutors in this role.)

The long duration of Problem 2 would require considerable student discipline if students were to be expected to work steadily, against weekly deadlines in other subjects. We provide structure to the problem with deadlines for the following stages:

- group submission of structural and functional prototypes and set of acceptance tests;
- individual submission of a significant piece of code which contributes to the group system where a pass on this stage is required for the student to earn the group marks of their group;
- a final individual submission of code which gives a large part of the marks for a student's practical assessment;
- group submission of the code for the system.

In parallel with these, each student does additional small exercises of their own choice. Each week they set a plan for these and in the following week, assess their achievements.

In theory, a student in the PBL course could do a very similar sequence of tasks to those of a conventional course. Each week should have them doing an excercise in some aspect of the course. In parallel, they are working on the larger task. The difference between the PBL student and their counterpart in our conventional course is that the former *decides* what tasks to do. They may well get some assistance and guidance from the tutor in selecting a task from our resources or they may invent their own tasks. Students make comments like

I knew I could not write this loop for my problem until I had done a simple one to print the numbers 1 to 10. So I did that and then I could see how to start the loop for the problem.

The final reflective part of the semester is where students write reports on the problem solved by the group, present the demonstration, and write a reflective report on their learning, with their weekly plans as supporting evidence. Such 'reflective' experiences are considered important to deep learning and are a standard part of the PBL approach.

2.2. Semester 2: Introduction to Computer Science

The second semester has two tasks, each running half the semester. A common element in one of these is an outward focus on people ("users") while the other is technocentric. As in semester 1, we offer choices in the problems for each task, with care to offer tasks oriented to a range of interests, including business, life sciences, and engineering.

The outward-looking task has an information systems orientation. This makes it feasible to create the range of problems we want to offer. All involve inheritance as a fundamental aid for modeling the entities managed. Problems are also designed to ensure that students have practice in the language and technical issues of managing files.

This problem type makes it natural to incorporate ethical issues such as who might access or modify data. It also provides a perfect context for issues of scalability: so students need to assess the speed of the system as the amount of data grows.

Examples of the problem choices are: managing a biodiversity survey, managing information for an entertainment advisor, managing activities for Olympic participants, managing the data for a school timetable, and managing product inventory for a computer vendor. In forming groups, students are encouraged to find others keen to work on the same problem.

The second problem in intended to develop students' technical programming skills, especially recursion. Here we use parsing as the common theme in the set of problems offered. Each requires processing of some input whose format has been defined recursively. This makes a recursive implementation both natural and high pay off. At the same time, the task introduces the use of classes which do not correspond directly to physical entities. It also provides a good context for use of the Composite object-oriented design pattern.

Groups were re-arranged for this task. Choices of problems include: developing a spreadsheet including complex formulae defining cell values; a query interface (modeled on SQL) to tabular data; a pretty printer for a subset of the Blue language [10] used in the subject, an interpreter for a simple imperative language, or a compiler for that language.

2.3. Learning resources

We have developed a range of resources to support the learning. One important type of support is examples for a problem of the same character. This helps students see the possibilities for the problem, gain a better understanding of the issues and learn broad and detailed technical skills. Perhaps most important, this serves as a starting point when students are stuck.

We now describe some of these examples. For the first semester simulation problem, we provide one simulation for an ecosystem and another for a lift (elevator) simulation. The latter

in available online and is also included in the printed resource book so that it can be used in the tutorials. In addition, since this is the first large problem, the first semester devotes some tutorial and laboratory time to tasks which help students learn how to learn from examples. There are getting-started exploration activities as well as tasks in exploring the structure of a program and seeing how to get the big picture. In tutorials, activities involve study of the details of the printed examples, where this involves learning how to learn new programming techniques, idioms and language features from examples. The example supporting the semester 2 information systems task manages student grades for a university.

Online resources include a self-assessment site where students can try out tasks which are typical of the standard they might expect in the examinations. The site also has several example answers and assessment criteria. If they assess one of these answers, they can also see our assessment and an annotated form of the example.

We have written chapters in the text and resource books on technical aspects of programming. These run the gamut of language issues from data and control flow to inheritance, including when not to use it. There are broader technical chapters on aspects like code cliches and software engineering issues. We also provide articles on the other issues in problems, for example ethics and asymptotic run-time estimates. There are class time discussion questions on these issues as well as the pragmatics of additional skills required in the problem, for example preparing oral presentations and report writing.

Academic staff present weekly seminars on the major issues of the problem. For example, in the information systems problem, these deal with inheritance, information systems in the industry, persistant storage and scalability.

2.4. Assessment

Assessment has a critical role in any course. This and the role of criterion referenced assessment has been described by Biggs [3, page 68]:

> *Assessment in practice has two functions: to tell us whether or not the leaning has been successful, and in conveying to students what we want them to learn…*
>
> *In a criterion-referenced system, the objectives are embedded in the assessment tasks. So, if students focus on the assessment, they will be learning what the objectives say they should be learning*

We have been very careful in the design of our assessment. Essential elements of our approach are:

- criterion-based assessment so we state carefully what we want student work to demonstrate;
- equal marks for exam and practical work so that students see that we equally value these;
- within the practical work, equal marks for the individual and group work so that students see we value both equally and so they are motivated to perform both individually and as group member;
- group assessment for tasks associated with group activity, such as group planning, manage-

ment and coordination and the group demonstrations;

- individual assessment for code, according to the usual criteria for design, correctness, style and documentation;
- barriers in practical work where the individual must perform some tasks to an adequate standard before they may move on, and before they are entitled to the marks earned by their group;
- minimum performance requirements for both examination and practical work so that students cannot pass unless they achieve that minimum standard on both;
- generic aspects are assessed on the examination as well as in the practical work both because we want to assess learning in these areas and we want students to know we see them as important.

Much of our work in this area is relevant to any course. An example of the marking scheme for one part of the assessment is included in the Appendix.

2.5. Staff development

A move to PBL calls for a radical shift in the role of teaching staff. In a large first year class, this means that introducing PBL requires a significant investment in staff development. We have addressed this in four main ways:

- literature about PBL;
- staff development sessions;
- scripts for teaching staff;
- teaching mentors.

The first is the easiest approach. However, it has limited value. First, it is difficult for people to make the time to read yet another set of papers. More importantly, as we found in the early stages of using PBL, the mental shift required is hard to achieve from merely reading papers. It is too easy to fall back on previous teaching and learning experiences.

We have found that intensive staff development sessions provide an excellent starting point. We run these for three days, with mornings devoted to classroom sessions and afternoons for practical activities. The morning sessions are generally attended by most of the staff, including experienced PBL teachers. These people can be spread through the groups used for most sessions. Less experienced staff are encouraged to do additional work in the afternoons.

Once the semester starts, we continue to support staff with detailed scripts. These map out

- preparation tutors should do for their classes;
- an experienced teacher's assessment of the likely concerns students will have at this stage, especially where this relates to problems we know students will have with accepting the strangeness of PBL;

- tips for dealing with these problems;
- how to run each activity, in terms of how long each aspect should take, what students should achieve in that time, what to do if students are not progressing as the schedule says they should, how difficult students typically find the activity, what resources support the activity;
- explanations for our design of each of the activities since we need the commitment of the teaching staff and we need to be sure they appreciate the purpose of learning activities.

The first weeks of the first semester have very detailed scripts, up to eight pages long. As the week progress, the guidelines, suggest more choices for the teacher to make and there is less detail. By the end of the first semester, the outline for a week's activity is usually two pages long and in second semester, formal scripts are not provided. Staff feedback indicates that this level of support is appreciated, especially by those teaching for the first time.

The final support comes from structuring the course into sections, with an experienced and committed person as Section Leader. This person has weekly meetings with the four or five tutors in their section. This gives new staff the opportunity to discuss problems that they and their students are having. The group can discuss ways to deal with these.

3. Trial implementation

Since the shift from a conventional course to PBL is so radical, we first trialled PBL with a small group of students in 1996. Benefits of the trial were:

- careful evaluation of it was to inform the decision to move to PBL (or not);
- we needed an opportunity to learn how to run a PBL course and a trial group was better for this than a large class;
- when we were ready to move to full implementation of PBL, the trial provided solid answers to the questions of students and staff who were concerned about difficulties they encountered;
- and, although we had not anticipated it, its results helped us when we met challenges in the full implementation and we, ourselves, sometimes questioned the wisdom of PBL.

The trial involved an initial group of 42 students chosen randomly from a pool of volunteers; it was staffed by one of us and one regular CS1 tutor. It was structured similarly to the format we have described for the current course, although we have refined many details over the three years of full implementation.

Our trial implementation was complemented by an extensive evaluation undertaken by staff with professional expertise in the evaluation of higher education courses. Here we address three issues:

- assurance that the PBL students were not less competent at programming than main group students;

- evidence that the additional generic skills that PBL is designed for were in fact being learnt;
- data about students' attitudes and perceptions towards PBL, since they can have a marked effect on learning.

For ethical reasons it was necessary to select the PBL stream from volunteers, so we cannot rule out self-selection bias altogether. However, we can say that the PBL group is typical of the main group in ability, as measured by the Tertiary Entrance Rank (TER), a single number summarising the performance of each student on the university entrance examinations. For the main group the average TER was 80.9 with standard deviation 9.2. For the PBL group the average TER was 81.2 with standard deviation 9.6, a difference that was not statistically significant at the 5% level.

It was necessary to use Pascal in the trial since the students needed competence in it when entering second year, so the expected synergy between problem-based learning and object-oriented programming could not be assessed. On the other hand, students in both groups sat the same examinations, and these examination results provide an excellent basis for an objective comparison of the outcomes of the two groups.

3.1. Outcomes of the first semester

Our analysis of the results of the first semester examination found no significant difference between the performance of the PBL group and either a group from the main CS1 class matched on the background variables of academic achievement and previous computing background, or the main class as a whole. When the results of individual examination questions were analysed, one question was answered significantly better by main group students (not surprisingly, since that question related to a specific way of drawing data structures, as used in lectures and this was not used in the PBL stream) and one question was answered significantly better by PBL students (for no clear reason). These results effectively answer concerns that eliminating lectures will reduce student learning.

We note that the examination was set by the lecturers of the main CS1 class. As one would expect, this meant that the particular approaches of those lecturers was reflected in the questions set. That the PBL students achieved the same levels of performance as the main CS1 class is a quite positive outcome for the PBL trial.

On the second issue, the learning of generic skills, we do not have quantitative comparisons between the main and PBL groups; but in many cases there is no basis for such comparisons. PBL students were required to design, plan, implement, test, manage, and report on a large group software project. No comparable demands were made of students in the main group. This is a crucial advantage of PBL: that without losing any of the technical skills, it makes room in the course for activities that encourage generic skills. We would have liked to assess the generic problem solving skills, as for example, has been done in the case of an introductory engineering course [15]. Our limited resources made this infeasible.

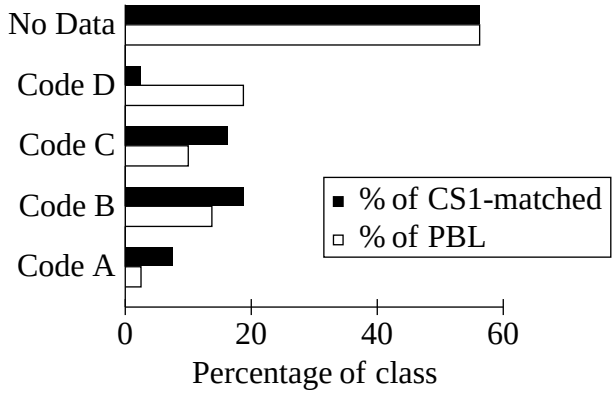
To address the third issue, student perceptions and attitudes, we conducted an open-ended questionnaire at the end of the first semester, in which we asked students to complete the statement 'After one semester of computer science my attitude to this course is …'. Answers were coded on two scales: how the students felt they were engaged in meaningful learning, and their emotional response. The answers were analysed by a graduate student from the Faculty of

Education at our university, under the direction of the author from that faculty. The PBL students were compared against the matched group from the main CS1 class.

To analyse the students' responses from the point of view of learning attitudes, the responses were coded in the following classes:

Class	*Example*
A. No meaningful learning	"I feel I have great difficulties in learning this course. I consider the speed of everything too fast. I might drop out."
B. As A but hopeful	"The work load is very heavy and I feel disheartened but I suppose if I kept at it, I'd find it a lot easier to cope."
C. As A but improving	"It is pretty good. I find it difficult sometimes but once I have worked it out it is enjoyable."
D. Meaningful	"Learning all the time! The best and most enjoyable practical course I have enrolled in so far!"

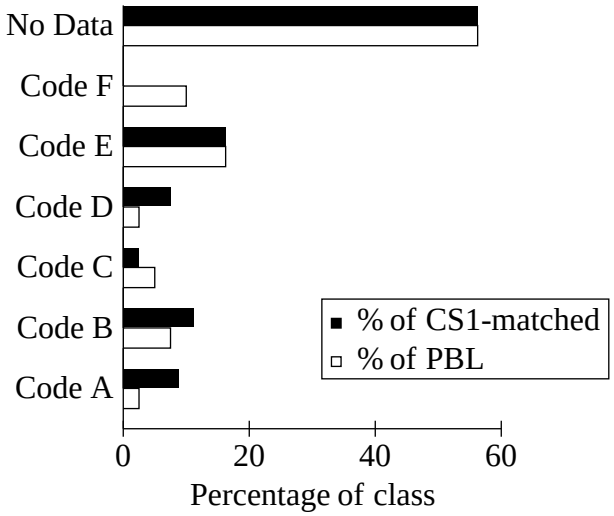
Conclusions must be cautious because of the high percentage of students who failed to respond. However, as indicated by the graph below, the largest proportion of PBL students who answered were in the meaningful learner category:



The second analysis coded the responses for their emotional content:

Class	*Example*
A. Despair	"I feel very depressed and anxious about this course." "I'm sure I'm going to fail and there isn't anything I can do about it."
B. Dislike	"I do not like computer science." "Disheartened, it wasn't what I expected and I can't imagine working with computers any more."
C. Bored	"It is very boring." "This course is very slow and dull."
D. Hopeful	"Sometimes I feel overwhelmed by the work, but I feel better when something goes right."
E. Positive	"Positive, I think I'm going to make it." "It's good overall, I feel better when everything is going well."
F. Great	"I love this course, it is my favourite." "This course is far more varied and enjoyable than I would have imagined."

The results are



Once again, we need to note the large number of non-responses. For the others, we can see that the more positive emotions are expressed by larger proportions of the PBL group. Similarly, smaller proportions of the PBL students expressed negative emotional responses.

In interpreting these comparisons it must be remembered that the PBL course was a first prototype. It suffered from several teething troubles that did not afflict the mature, established course. On the other hand, staff enthusiasm for the new course was very high.

3.2. Longer term follow up

Funding for the detailed analysis of the trial covered the first semester. So the most detailed analyses are for that semester. We also compared the results for the second semester practical examination. This was a three-hour examination during which the students worked individually at a computer terminal on a previously unseen programming problem. Students who failed the examination had an opportunity to try again later.

The percentage of students passing the first sitting of the practical examination was 48.5 for

the PBL group (with average marks of 55% and standard deviation of 40.5%), and 48.1 for the main group (with average marks of 58% and standard deviation of 38%). The differences are not statistically significant at the 5% level.

We analysed the longer term performance of the students in the PBL trial. Of the 42 students who began first year, 22 completed second year studies and 16 studied third year level subjects. With such a small sample, one needs care in interpreting statistical comparisons with the main class.

3.3. Examination performance and continuation rates

Analysis of examination results shows no significant difference between the trial group and the main class.

A similar study of the continuation rates for the two groups indicate a slightly higher retention to second year level for the PBL group and then a slightly lower level for the PBL retention to third year level. This might be consistent with a positive first year experience, followed by a more negative perception of the second year units. These were all taught in a conventional format. Perhaps more importantly, at second year level, the PBL trial group were part of a large class dominated by those who had studied in the conventional format. Lecturers would have have tended to be most responsive to the bulk of the class, perhaps being less aware of those from the PBL trial group.

3.4. Affective measures

In 1999, we undertook a limited study of the long term affective aspects. Two groups of students were invited to complete an open-ended questionnaire. One group was the thirteen students who began first year in 1996 and were in the fourth year Honours class in 1999. This included two students from the PBL trial group. In addition, 5 had been in the first semester Advanced class which was in PBL format. The remaining 6 had been in the conventional course. The second group approached was those students from the 1996 first year class and still studying third year level units in 1999. These students had taken more than the minimum time to reach third year level. This may be due to a range of reasons. For example, some studied in combined degrees where it is normal to study third year level subjects in the fourth year of study. Others had failed subjects or taken time away from their studies. All such students were mailed. Responses were collected from 6 students, 4 from the conventional course, 1 from the PBL trial group and 1 who had been in the Advanced first semester course in PBL format. We cannot claim these groups as representative of the class as a whole. However they are diverse. Also, the students who study in the Honours class are important as future researchers and top students.

The questionnaire asked open ended questions like: ‘What do you remember about your first year of Computer Science’. ‘What do you think were the positive aspects?’ ‘What do you think were the negative aspects?’

There were clear differences in the character of the answers from those who had studied in PBL format compared with those in the conventional course. For those in the conventional course, answers focused on the specific language and lectures.

I remember learning Pascal. That is not to say I remember Pascal

Fairly well paced except for tracing by XXX, pointers by XXX and files by XXX

They recalled the positive aspects in the same terms, citings aspects like learning programming skills as a positive experience. Negative aspects were varied but involved machine resources and specific lecturers. However, one recalled that they "didn't get to know many people".

The students who had studied in the PBL format had broader answers including aspects like "friendship" and "group discussion" as well as working on "challenging problems", "self-directed learning" and the "group work". Some commented on the self-paced and self-directed learning as a positive aspect. For example,

learning things by myself is much better than being spoon fed by lecturers.

Another recalled the positive aspects as:

anything PBL …. it was the only stuff I remembered and it was fun

it really was a lot of fun …. it was a rapid introduction to programming and more was learnt in compSci than in any of my other subjects

Group work was cited with both positive and negative comments. Students liked the small, helpful and friendly environment of PBL. Negative comments included

difficult due to lazy team members

[being] held back/slowed down by the group

On the other hand, one member of the trial group had extremely negative memories, with comments like:

we were taught nothing and had to learn everything about Pascal ourselves … it was too much of a burden esp for people who came to uni with no previous programming experience nor knowledge of a computer

without no one to even teach you the basics at least, you gave up after a while … a hopeless situation

whatever I learnt I learn from other students or learning myself.

and this student flatly stated that there was nothing positive about their first year computer science experience.

Many PBL courses report a minority of students expressing this view. Such comments correctly identify some aspects of PBL. Unfortunately, they also indicate a failure to help students see the advantages of this style of learning. They also indicate the need for better scaffolding for students.

Negative aspects of PBL emphasised the lack of guidelines and feedback, lack of structure and the uncertainty as to what exactly was expected. Other comments included having problems with the structured exam.

A largely consistent picture emerges for the students who had studied in the semester 1 PBL format Advanced class. They liked the intellectual challenges and open-endedness of PBL.

They identified group work as useful and pleasurable but also as a source of frustration.

4. Experiences from full implementation

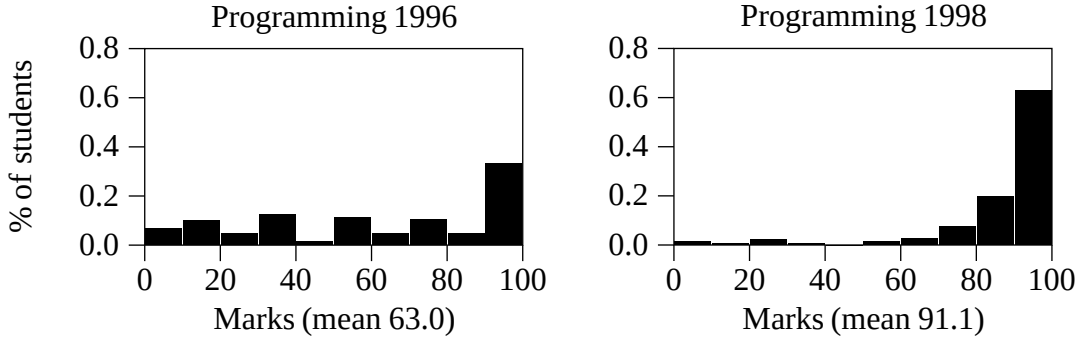
In three years of full implementation, we have refined the courses considerably. One would have expected this with any new courses, even ones which only involved a move to a new programming language. However, with the radical shifts involved with the move to PBL, there has been more learning on our part.

For example, the assessment criteria are particularly important in our PBL courses. A conventional course can get by with vague assessment criteria, although it is wise for every course to make best use of clear assessment criteria to help communicate the learning outcomes required. In PBL, with the very general problem statements as a starting point, it is all the more critical to give students a very clear understanding of how they will be assessed. One of the well acknowledged problems that students report in PBL is that they are unsure how much they need to know, how far to explore their problems and when they can reasonably stop learning. We have addressed this difficulty with a carefully crafted set of assessment requirements. We continue to improve aspects like this as we gain experience.

At this stage, we have evaluated several aspects of the courses. We now report these.

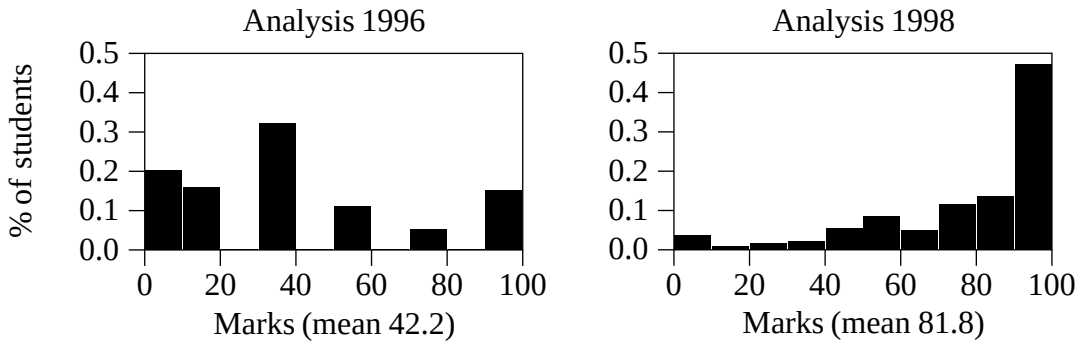
4.1. Learning outcomes

The following sets of graphs compare the results of second-semester examinations in the last non-PBL year (1996) with corresponding results in the second full PBL year (1998). There has been a substantial improvement in basic programming competence:

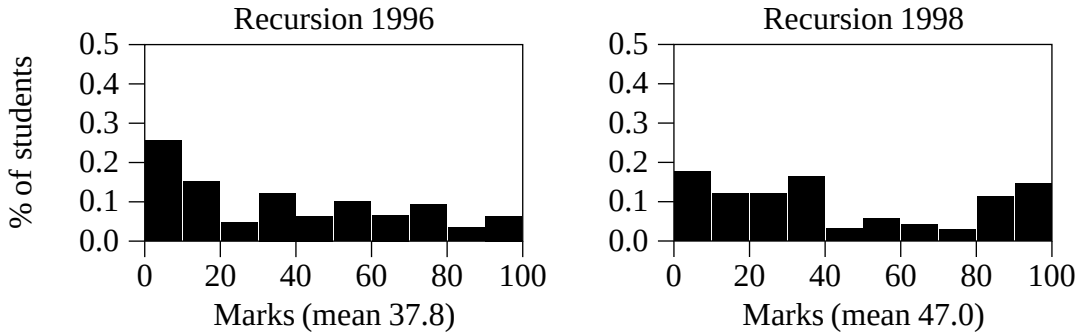


The two questions are list traversal problems; the questions are different but they are of similar difficulty. We believe that this improvement is partly due to our new software, the Blue programming environment [12] Cite $kolling1999. It is also partly due to the new curriculum, which returned programming to the centre of the second semester unit of study.

Improvements have also occurred in questions on topics often thought to be beyond the grasp of the average student, such as time complexity analysis:



These questions again are of similar difficulty. Another advanced topic, and a key skill for students progressing to second year, is recursion:



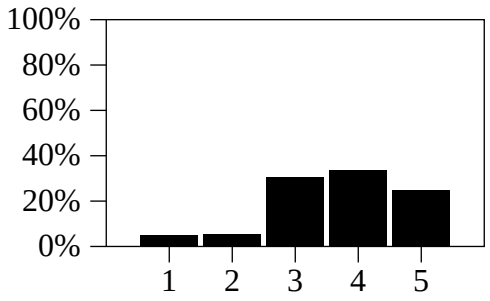
Although we clearly have more work to do here, the 1998 recursion question asks the students to write a recursive descent parser, and thus is considerably more difficult than the tree traversal question from 1996. These improvements have occurred because these topics are now integrated into the second semester problems, whereas in the old unit they were buried in lectures poorly attended and perceived as irrelevant by many students.

4.2. Affective aspects

Each year, we survey students in the middle and end of the first semester course and at the end of the second semester course. Students indicate satisfaction with most aspects. The seminars consistently rate less well than tutorials and workshops. We have revised the content and style of seminars over the years and student satisfaction has steadily improved.

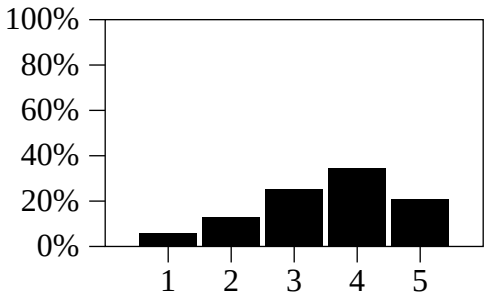
Surveys have been used to inform refinement of the course and they have indicated a steady improvement in student responses. Here are some typical response summaries for 1999. The scale is 1 = very low, 2 = low, 3 = average, 4 = high, and 5 = very high:

How much did you learn in this course?



Responses: 461; average: 3.7

How interesting did you find this course?



Responses: 460; average: 3.5

Looking through the free text responses, only a handful of respondents are clearly unhappy with PBL or other aspects. Many students gave constructive suggestions. When asked 'What was the best thing about this course?', the most common responses were: group work; PBL; open-endedness and realism and challenge of the work; learning to program; object-oriented programming. When asked 'How can this course be improved?', the most common responses were: more and faster terminals; change to a real world language; lighten workload, especially early on; do something about group members who do not do their fair share.

Tutor feedback has indicated even further improvement in the unit. For example, one tutor who has taught since 1997 commented:

> *Overall I felt this year went better than previous years. I saw students who were using Files, Lists, HashTables and all kinds of things this time around, but last year saw very little. I also saw a lot of code re-use. Students in my class were pulling apart Michael Koelling's Earth simulator, and Gary Capell's elevator simulator and learning heaps from them.*
>
> *The most important areas for improvement are the management of the process aspects, ensuring that students work more steadily and reflect on their learning.*
>
> *The new criterion-based assessment needs tuning.*

5. Related work

Our adaptation of a quite pure form of PBL to teaching foundation computer science courses seems to be novel. However, there are many elements of other computer science courses which share some features of our approach.

For example, about ten years ago, there was an innovative problem-based learning course in Bachelor of Informatics degree at Griffith University [13]. It had a very different emphasis, with the social context of computing being equal to the technical core. With time, this course ended, due to various difficulties, including the departure of some key staff members and antipathy to PBL by others. We have tried to learn from this effort. Our trial was important for building staff support. We have also devoted considerable effort to staff development both to improve the teaching and to improve staff acceptance.

We have also been able to learn from the applications of PBL to Engineering programmes

such as Woods [17, 18, 19] and more recent cases like Reeves and Laffey [15]. Although much of that work is not directly applicable, it is closer to our situation that the larger scale use of PBL in the medical and paramedical areas.

There has also been a Computer Science course taught with success in an Aeronautical Engineering university [7]. In that case, the relative consistency in the student population makes the course design properly directed at the needs of aeronautical engineering. This appears to have helped students appreciate the usefulness of this material.

More senior level courses commonly have various degrees of PBL flavour. In particular, most Computer Science majors do a senior level course in what is often called Software Engineering or may be called a capstone project. We emphasise that a basic principle underlying such courses is that the students have learned the basic technical skills needed for the project task: the goal of the course is to give them the opportunity to gain practice in integrating those skills learned in various courses. This is quite different from PBL where the problem is the driving force for new integrated learning, not just the integration of existing knowledge. In practice, of course, we find that such courses do involve considerable learning of new skills required for the task, as well as building on existing knowledge. Such courses typically require and develop generic skills in communication and problem solving, though the explicit support of that learning may not be provided. In practice, if not in philosophy, these senior courses have a strong PBL flavour.

Artificial Intelligence has also been taught using problem-based learning [14]. Several other recent initiatives in Computer Science education [2, 5, 6, 9, 16] have elements of problem-based learning. To our knowledge, however, we are the first group to create an entirely problem-based first year Computer Science course.

6. Conclusions

We adopted problem-based learning, not for its own sake, but because we saw that it would foster the following goals.

An integrated curriculum. The educational literature warns against compartmentalized units of study that produce students who cannot integrate the different parts of their knowledge. Although a fully integrated degree was beyond our scope, the earlier conventional foundation courses had compartments that bore out the literature's predictions. The new courses are fully integrated, since students bring all their knowledge to bear on solving a few large problems.

Competence in computer programming, using a modern object-oriented programming language and environment. This is our main technical goal and the one students are most motivated to achieve. We have made this goal central by structuring the units as a sequence of large programming problems, and requiring our students to use the 'Blue' programming language and environment.

Ability to enhance program quality by using formal methods. This goal suffered from compartmentalization in the old units. We have integrated it with programming by adopting a programming language with features to support it, using them routinely ourselves, and requiring our students to do so.

Ability to analyse the running time of programs, and to produce programs with low running times. This was another victim of compartmentalization in the old units. One of our problems

requires handling large quantities of data, and so students must master these techniques if their programs are to be efficient. We support their learning with tutorial exercises, a text book chapter, and a seminar.

Ability to use recursion. Recursion is an important conceptual tool and programming technique, and traditionally a difficult topic for novice programmers. One of our problems is inherently recursive, so that students must master recursion in order to solve it. We support their learning with several text book chapters, seminars, and tutorials.

Independence and initiative. These are important generic goals because the old units were widely criticized for stifling them. The problems we offer are loosely specified, leaving plenty of scope for student initiative. Students discover they can learn by themselves, using a range of resources. They are aided in learning to do this because of the PBL planning structures and tutorials which develop metacognitive skills.

Critical thinking and problem solving. These are crucial to effective programming, especially at the higher levels of analysis and design. We ensure that students encounter these higher levels by having large projects requiring careful analysis and design. One of our tutors' most important roles is to guide students in learning these skills as they work on the problems. We also provide tutorials to help students develop generic problem solving skills.

The ability to work in a group. This is important for employers; successful groups also increase students' confidence and initiative. All our problems after the first four weeks are group problems. We support groups by identifying specific roles for group members, providing class time and guidelines on group management, monitoring group planning and progress and assigning marks for group management and reflection on group processes.

Communication skills. This is another goal highly valued by employers. Students working in a group naturally learn to communicate with one another. At the end of each problem the students give a demonstration, during which each student must speak, and a written report. We also require appropriate English commentary within the students' programs.

Planning. Real-world programming projects are very large, and most failures are due to poor management rather than technical problems. To expose our students to these issues, we set large problems (7–10 weeks' work by a group of four or more students) and assess group and individual planning as well as product.

PBL fosters generic skills such as group work - also necessary for a full appreciation of object-oriented programming - planning, problem solving, independent learning, research skills, writing, and oral presentation. These are University goals and also highly valued by employers in the computing industry. PBL allows students to achieve far more in their programming than was possible within the small, individual assignments of the old units; and it virtually eliminates plagiarism, a long-standing problem in the conventional course.

7. Acknowledgements

The introduction of Problem-Based Learning for our large first year classes required many important contributions. There have been University and Faculty grants: Teaching Quality Grant 1995; Student Progression Assistance Scheme Award 1997-8; and Information Technology Committee Grant 1998-9. Many departmental staff have been involved; particular support was provided by Professor John Rosenberg and Dr. Michael Kölling.

Appendix A. Examples of teaching materials

A.1. A criterion based marking scheme

The marking scheme below is from the first semester Resource Book, 1999. We actually use this marking scheme to do the marking, and it is there in the workbook for students to consult. Not only does this ensure that students are informed about our intentions, it acts as a powerful communicator of what it is that we value. It also communicates the qualitative difference between the different grades awarded.

Note that you must earn a Pass grade on Problem 1 in order to gain certification. (And you must have certification on Problem 1 before you can begin Problem 2.)

Criteria for a Pass in Problem 1

To earn a Pass grade, the submission must have all the following:

- It must begin with a signed statement that 'the code you have submitted was entirely written by you'. (If you needed a significant amount of help for any other aspect of the code, you should explain that near this signed statement.)
- Your code should be demonstrably able to do the job it is supposed to do: it must work.
- Your submission must include one whole class.
- Your code must make use of at least one loop.
- It must make use of at least one if-statement.
- It must make of at least one LList.
- Each class must have a comment stating what it does. (It must actually do this correctly to be judged working and the comment should describe accurately what it does)
- You must submit a Test report which list the tests you did to convince yourself that it works correctly: maximum length is 100 words.
- In your 3-hr lab you must demonstrate that your code works by doing these tests (maximum time for this demo is 5 minutes).

Additional crieria for the grade Pass

At least half of the following must hold:

- The class interface should state the author and sources of significant aspects (for example, if it is based on information from an accounting book, it should give the reference)
- The class and each routine should have comments explaining what they do.
- Each routine should have comments explaining what they do. It is usually a good thing to explain each routine in terms of its parameters. (eg 'deposit' accepts an 'amount' in dollars

which is deposited in the account, with a deposit fee deducted.)

- Names for classes, routines, parameters and variables should be helpful (eg Account is a good name for a bank account class, Snazzi is not.) Choice of identifiers should follow the style of examples in the textbook and in the class directories.
- There is an attempt to write preconditions and postconditions.
- Layout must be consistent and clear, with indentation showing code structure.
- There should be helpful comments through the code as needed.

Criteria for a Credit

All the required and additional criteria for a Pass plus most of the following:

- All the Blue control structures should be appropriate for the task.
- There should not be tedious code (for example, it is a bad idea to use 30 print statements if you need 30 lines, each with the same string - as a nascent computer scientist you must sense that there has to be a better way to do this, and be determined to find it.)
- This aspect means that presentation of the code means the reader can understand it with minimal effort.
- You have acknowledged sources and resources which informed your work on the code. Although you should have written the class yourself, you will not have done it in a vacuum. For example, you might have modelled your code on some of the examples in the text or in our examples. You might have used various library classes, like LList or random.
- Testing is convincing (within 100 word limit), stating: the purpose of the test; the input for the test; the expected output or behaviour; observed behaviour. A tabular presentation is probably a good idea.

Criteria for a Distinction ++

All the required and additional criteria for a Pass and Credit plus most of the following:

- Code should be clear and simple.
- Each routine should do a well defined task the same one described in its interface comment), have good choice of parameters and good identifiers.
- Code does something interesting and challenging.
- More sophisticated Blue aspects used (eg nested loops, more than a single LList)
- Testing in report and demo are minimal and elegant.
- Each test should test a different aspect of the class and the ‘purpose of the test’ should make this clear.

A.2. A concept inventory

Task A Concept Inventory

Use this inventory to summarize what you have learnt in Task A, and to find gaps in your knowledge compared with what is expected. You migh also wish to review the concept inventory in the Problem 1 work book from COMP 1001. Score yourself against each item like this:

1. I've never heard of it
2. I've heard of it but know nothing more than that
3. I know this well enough to try to apply it
4. I know this and I can apply it
5. I know this well enough to explain it to a friend

	Score	*Notes on what to do about this item*
Inheritance		
Why inheritance is needed		
How to get inheritance		
Deferred and redefined routines		
Polymorphism		
Static and dynamic type		
Assignment attempt (?=)		
Inheritance and creation		
Inheritance of contracts		
super		
File handling		
The lifetime of routines, objects, files		
TextFileHandle and *FileSysHandle*		
Scalability of software		
Why scalability is important		
Worst case and average case		
O-notation		
Calculations with *O*-notation		
$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$		
Analysing routines		
Effect of algorithms and library classes		
Ethical awareness		
Identifying users and ethical issues		
Effect of ethics on software design		

34

Page 34 of the COMP 1002 Workbook for July Semester 1998, showing the Task A concept inventory, a checklist that students can use to ensure that they have covered all the concepts we expect them to by the end of Task A. For example, few students would have encountered the ?= symbol, and this inventory can act as an indication to them that they should find and study it.

References

[1] M. A. Albanese and S. A. Mitchell. Problem based learning: a review of literature on its outcomes and implementation issues. *Academic Medicine* **68**, 52–81.

[2] Catherine C. Bareiss. A semester project for CS1. In *Proceedings 27th SIGCSE Technical Symposium on Computer Science Education*, pages 310–314, Philadelphia, Pennsylvania, 1996.

[3] J. Biggs. What the Student Does: teaching for enhanced learning. *Higher Education Research and Development* **18** (1), 57-75 (1999).

[4] D. Boud and G. Feletti. *The Challenge of Problem Based Learning.* Kogan Page, 1991.

[5] Curtis A. Carver, Richard A. Howard, and William D. Lane. A methodology for active, student controlled learning: motivating our weakest students. In *Proceedings 27th SIGCSE Technical Symposium on Computer Science Education*, pages 195–199, Philadelphia, Pennsylvania, 1996.

[6] Suzanne W. Dietrich and Susan D. Urban. Database theory in practice: learning from cooperative group projects. In *Proceedings 27th SIGCSE Technical Symposium on Computer Science Education*, pages 112–116, Philadelphia, Pennsylvania, 1996.

[7] I. Hirmanpour, T. Hilburn, and A. Kornecki. A domain centered curriculum: an alternative approach to computing curriculum. In *Proceedings of the 26th SIGCSE Technical Symposium*, pages 126–130, 1995.

[8] J. Kay, J. Lublin, G. Poiner, and M. Prosser. Not even well begun: women in computing courses. *Higher Education* **18**, 511–527 (1989).

[9] Raymond P. Kirsch. Teaching OLE automation: a problem based learning approach. In *Proceedings 27th SIGCSE Technical Symposium on Computer Science Education*, pages 68–72, Philadelphia, Pennsylvania, 1996.

[10] M. Kölling and J. Rosenberg. Blue – a language for teaching object-oriented programming. In *SIGCSE Technical Symposium on Computer Science Education*, 1996.

[11] M. Kölling. Teaching Object Orientation with the Blue Environment. *Journal of Object-Oriented Programming* **12** (2), 14–23 (1999).

[12] M. Kölling. The Blue Language. *Journal of Object-Oriented Programming* **12** (1), 10–17 (1999).

[13] S. E. Little and D. Margetson. A project-based approach to information systems design for undergraduates. *Australian Computer Journal* **21**, 130–138 (1989).

[14] L. Cavedon, J. Harland and L. Padgham. Problem based learning with technological support in an AI subject: description and evaluation. In *Proceedings of the Second Australian Conference on Computer Science Education Conference*, pages 191–200, Melbourne, Australia, 1997.

[15] T. C. Reeves and J M Laffey. Design, Assessment, and Evaluation of a Problem-based learning Environment in Undergraduate Engineering. *Higher Education research and Development* **18** (2), 233–246 (1999).

[16] Massood Towhidnejad and James R. Aman. Software engineering emphasis in advanced classes. In *Proceedings 27th SIGCSE Technical Symposium on Computer Science Education*, pages 210–213, Philadelphia, Pennsylvania, 1996.

[17] D. R. Woods, J. D. Wright, T. W. Hoffman, R. K. Swartman, and I. D. Doig. Teaching problem solving skills. *Engineering Education* **66**, 238–243 (1975).

[18] D. R. Woods and R. J. Sawchuk. Fundamentals of chemical engineering education. *Chemical Engineering Education*, 80–85 (1993).

[19] D. R. Woods. *Problem-Based Learning: How to Gain the Most from PBL*. McMaster University Bookshop, Hamilton, Ontario, 1994.