1. Introduction

Welcome to the course. And congratulations on making a good subject selection decision. We’re not biased, of course... Just very well-informed...

First year (1000-series) courses have an emphasis on developing the essential skills that form the foundation for all other courses. These skills include problem-solving, design, technical and programming abilities, as well as the ability to effectively use appropriate technology. In second year (the 2000-series courses) and third year (3000-series courses) these skills are used to explore exciting areas of computer science. A fourth year (honours) is available for students who have demonstrated the required level of proficiency, and involves research experience in a wide range of areas reflecting the diverse interests of academic staff in the department; the honours year may be taken full-time or part-time (with the possibility of tutoring work in the department).

We know that many of you will become interested in pursuing research in one or more of the many interesting research areas available in computer science, and encourage you to think of this possibility as you proceed through your undergraduate studies with us. You may develop specialised interests which can form the basis of future honours, masters or doctoral studies in the department. This may seem a long way down the track, but the seed is being planted with your enrolment in this course.

In order to place COMP-1001/1901 in its proper context, a very brief introduction to other courses offered by the department follows, after which an introduction to COMP-1001/1901 is presented.

1.1 Overview of other CS courses

For more detailed information on available courses, please refer to the departmental handbook and your faculty handbook. What follows is a sweeping statement of available courses, presented as a means of placing the current course in context.

1000-series

Of immediate relevance is the fact that following successful completion of this semester’s course (COMP-1001/1901, "Introductory Programming") you have two second semester courses available to you.

- COMP-1002/1902 ("Introductory Computer Science") extends the development of your programming expertise as well as completing your familiarisation with object-oriented design features, and prepares you for entry to intermediate computer science courses by introducing concepts of correctness, reasoning about programs, efficiency, and other facets of computer science.

- COMP-1000 ("Information Technology Tools") is designed to develop students’ expertise in use of a wide range of important classes of applications which permeate the workplace and home computing environments. Tools such as web-browsers, spreadsheets, databases, word-processors, etc., are examined as independent packages, and also in terms of their integrated use, and the incorporation of such packages as problem-solving tools.

Important: If you intend to do intermediate Computer Science (and who wouldn’t, hmm?) then it is a requirement that you complete COMP-1002/1902. The nature of COMP-1002/1902 is, in part, to specifically prepare you for these courses by bridging programming and the wider discipline of Computer Science. You may also enrol in COMP-1000 to gain exposure to a different set of pragmatic skills. These courses (COMP-1002/1902 and COMP-1000) are designed to complement each other, and individually address quite different skills.

If you do not intend to enrol in intermediate Computer Science then you should consider enrolling in COMP-1000 to provide you with a set of skills which are highly regarded in the workplace and may also prove useful in your other studies. On the other hand, you may still want to enrol in COMP-1002/1902 to extend your object-oriented programming and design skills to their completion.
2000-series

Some of the offerings of the intermediate-level computer science courses include:

- the organisation and functionality of computer hardware;
- the representation of data within a machine;
- low-level systems software, including the nature of compilers;
- important data structures and their use (B-trees, heaps, hash tables, adjacency matrices, etc);
- languages and logic;
- programming in industrial-strength languages including Java and C++;
- use of the UNIX operating system, shell scripts, and programming tools.

Details can be found in departmental or faculty handbooks.

3000-series

Some of the offerings of the senior-level computer science courses include:

- design and analysis of algorithms;
- programming games with artificial intelligence;
- knowledge representation and expert systems;
- construction of simple computers;
- application of computer graphics algorithms;
- introduction to database management systems;
- logic programming with the language Prolog;
- computer connectivity, networking, and internetworking;
- writing simple client-server programs in C;
- object-oriented design and productivity tools;
- the design of operating systems;
- software engineering;
- the design and creation of good user interfaces, including Python and TK Toolkit;
- programming projects with a variety of different emphases.

Details can be found in departmental or faculty handbooks.

Honours

The department has a diversity of expertise and professional interests. This is evident in the variety of offerings which are made every year for honours students. Students work with a member of staff on a research task, resulting in the production of a thesis. In addition, they do some advanced topics which vary from year to year.

Patterns of Study

What isn’t obvious from the very brief overview presented above is how courses in your degree may be structured to develop specialist expertise. Consult the handbook for details of suggested course structuring leading to specialised skills in areas of:
• network management and systems administration;
• embedded systems programming;
• commercial programming;
• scientific programming;
• bioinformatics specialisation;
• geographic information systems programming; and
• specialties in mathematics, software engineering, etc.

1.2 About the Introductory Programming course Comp1001/1901
This course focuses on the development of programming and problem skills, cast in terms of essential principles of software engineering. There are four important and innovative elements of the course.

1. It is offered in a Problem-Based Learning (PBL) style. This provides an authentic group-based programming environment based on student-centred approaches to learning. Refer to the chapter on PBL for details.

2. It uses an object-oriented language called "Blue". The language was developed within the department to maximise the learning of important concepts in object-oriented programming; an intuitive environment assists in rapidly developing a "feel" for the development of code. Refer to the chapter on Blue, and the "Readings" section for further details.

3. It has a layered and sectional administrative structure described in Chapter 2. This gives staff a greater autonomy and flexibility in course development and delivery. It also means that students will mainly work as part of a class group of around 20-25 students. The whole section has 5 or so such class groups is co-ordinated by a section leader who is responsible for about 125 or so students.

4. It offers a choice of "flavours" of programming tasks within the PBL mode of learning. You may choose (dependent, as always, on logistical matters such as timetabling) between a number of different programming problems from week 4 of the semester.

Outline of Course Philosophy
This OO-programming foundation course introduces a software engineering view of the process of constructing programs, with code reuse and effective application of a small class library, concern for good design-in-the-small, systematic implementation, code quality, thoughtful testing and appreciation of high quality programming as a challenging human activity.

The course aims to develop problem solving skills that are fundamental to effective programming: learning is driven by large problems that students tackle in groups. Various activities help students to learn how to formulate goals, identify strategies for achieving these, find and use resources, evaluate their progress towards a goal (especially learning goals) communicate the processes orally and in writing and co-operate with groups members.

1.3 Why Blue?
Blue is a new language. In using Blue, we have decided to use a very modern system. Blue is, in some areas, still under development. This presents many challenges and opportunities. The biggest advantage, of course, is that we are convinced that in using Blue you are well-positioned to get the best programming education possible. But there are also other aspects.

One of the great opportunities is that you can have important input in the evaluation and modification of the
system. Unlike with other programming environments you can buy, you are not stuck with what we have got. If during the course of semester we can together think of improvements that make our system better, easier, or more fun to use, then we have the chance to incorporate those changes.

There has been a lot of interest in the Blue system from other universities, both within Australia and overseas. There is a good chance that other universities will adopt Blue as their teaching language. But you are in the unique position to be the ones that are at the forefront of the development, to give direct feedback and to be in close contact with the researchers who developed the system.

You are encouraged to become active in this development of Blue. Suggestions which are acted on, and new bugs which are found, will be credited to the first person to forward comments. Blue is currently available to run on many Windows95 and most Linux systems.

We use Blue in our introductory courses because we believe it supports learning of important concepts. On this foundation, second year subjects introduce the messier and more complex languages used in industry, like Java and C++.

1.4 Why PBL?

Nothing in education is so astonishing as the amount of ignorance it accumulates in the form of inert facts.
--Henry Brooks Adams

Knowledge that cannot be activated, integrated, or applied, but sits around in a state of dormancy hoping to awakened by a passing exam question is not the sort of knowledge valued in this course. It may be very good for intending quiz champions, however, so let us not dismiss it entirely. In an effort to encourage vital and dynamic education (of the kind that would escape Henry Brooks Adams’ criticism) we have incorporated Problem-Based Learning into our first-year programme.

Learning in PBL is characterised by:

- **Learning in Context:**
  Skills (such as programming) are learnt in order to aid in the resolution of a wider problem. In this it closely resembles the "real-world" situations in which those skills would normally be required.

- **Problems Motivate the Learning:**
  Unlike conventional approaches to education, in which you are told what to learn and then expected to solve a (usually meaningless) problem in order to test your recollection of the information, PBL presents students with an authentic problem which motivates their learning as needed to address the issues which the problem generates.

- **Integrated Learning:**
  As learning is motivated by the problem it will therefore not be limited to a rigid curriculum.

- **"Ownership" of the Problem:**
  There is usually more than one pathway to a goal. PBL acknowledges this in allowing learners to give their own meaning to the problem, and make choices based on this. Teachers serve to guide and facilitate, but the ownership of the problem is given to the students.

- **Self-Directed Learning:**
  Students will be largely responsible for the learning in which they engage. The problem (plus some additional small tasks, seminars, and self-assessment tasks) will ensure that the required content of COMP-1001/1901 is covered; however, students will navigate through the material on their own motivation.

- **Learning about Learning:**
  PBL also focuses on the learning process itself. As well as learning through problem-solving, students will be asked to reflect on the process of learning about problem solving.
• **Collaborative Work:**
PBL takes advantage of the well-known benefits of group-based work to encourage a stimulating environment for learning.

• **Ill-Specified Problems:**
Problems are authentic in that they are not artificially constrained to fit into a small design space. That is, there is not necessarily a single "correct" answer to the problem. The problem is left open to interpretation, and is expected to be redefined in accordance with a growing familiarity with it.

• **Recognition of Prior Learning:**
Students are not blank slates (I assume you agree), and enter a course with a variety of skills, experiences, and conceptions which are brought into play when actively engaged in learning in a PBL environment. There is often a circularity here; while these are valued in the learning environment, they are also often modified by ongoing experiences in it (hence the learning!).

**Opinions about PBL from Industry**
As potential programmers and computer scientists of the future, you might be interested in some of the feedback that came from figures in industry after reading our report recommending the change to PBL.

• I thoroughly support the change to PBL. I believe that theory and concepts are much better understood when they are thoroughly grounded.

• I feel that this is a great way of learning design and implementation issues that others have experienced.

• As a practitioner I really support the problem based approach. It appears to be much more relevant to the real world both in terms of the content and processes involved. It also seems to address the need for students to learn group and communication skills.

**1.5 Words of Wisdom!**
**Students’ Thoughts on Problem-Based Learning**
Collected here are some of the thoughts of previous PBL students. First are some opinions about PBL as a means of learning computer science; it is hoped that you will detect a sense of excitement about the approach from their comments. These were mostly collected from anonymous surveys, and may therefore be regarded as a legitimate comment on student reactions to PBL, and an indication of what lies ahead for you.

• I believe that I have greatly expanded my programming skills and knowledge during the course of this project... This came about mainly by being part of a responsible and mature group... and being able to work at my own level without hindering the overall progress of the group...

• I have found that this [project] has advanced my knowledge of Pascal and Computer Science considerably... [of course, now that we have changed languages from Pascal to Blue, we expect this to remain true for Blue!]

• In a limited amount of time we learned and applied new-found knowledge to a project that is neat, efficient and very modular.

• The Problem-Based Learning process taught me a great many things about the nature of programming. First, it showed the whole process of the software cycle, instead of the more standard single application style of learning programming techniques... The project in
particular was great because it gave great scope to explore the kind of design structure that we thought was most applicable to the project, allowing the designers/programmers to use their own discretion to pursue different aspects of computing power in the programs....

• The combination of learning.... has resulted in a great growth in my programming ability and my attitude to learning. I hope that other computer scientists will the gain the same way that I have in future years.

• This was the first project that I have enjoyed doing since finger painting in kindy 14 years ago. I believe that the project was divided pretty evenly throughout the group according to our individual abilities. ... 

• ...most importantly, I have grasped how much easier programming is with these new skills, and how, after a bit of effort, things fall into place.

• Since the problem is to solve in using new concepts the solutions that we have to contribute are also a large number. Because of the variety of solutions, the problem leads us to organise and discuss to have a best solution.

Advice from PBL Students
We've presented you with some reasons for taking a PBL approach to learning Computer Science. However, it would be a wasted opportunity if some warnings and cautionary tales were not also included, and a collection of these follow. As these comments come from students who only recently experienced a similar educational approach to the one you are about to embark upon, these comments could form the most important part of this workbook. They are an essential guide to potential trouble-spots. Some people seem to think that everyone should, at some stage in their childhood, stick their finger in the fan or put their hand on the hot plate. We take a different view of education that says it needn’t be painful (it should be enjoyable)! By presenting the experiences of others we can prevent anyone who cares to take notice from suffering the same problems. Other comments take the form of advice based on positive experiences which worked.

Working in a Group: Co-operation, Responsibility, Allocation of Tasks

• I have learned about compromise and working together in a group, which is extremely different from working on your own. I learned that a collective effort is needed to do the best assignment possible.

• I honestly feel that .A. and .B. could have been a bit more helpful and that .A. could have turned up to more Friday classes.

• [Regarding allocation of tasks within the group] It has been suggested that in future someone be set aside ... simply to set pre- and post-conditions for the other programmers to abide by, and to make sure test data was available and that everything integrated well... Overall, a productive experience.

• There are things which I would have wanted to improve on perhaps in the future. These things included the nature of the ongoing design process. This means that I would have liked more of the group all involved in the ongoing design process, and the overseeing of the project. Or, alternatively, having some mutual consent in the group for some leadership in where the project was going.... Some more overall commitment to the project’s big structure would have allowed the project to finish earlier, testing to happen earlier, and would have allowed the aims of the group to be more accurately completed.

• All group members communicated at an incredibly functional level, with daily scheduled programming and discussion meetings, the use of on-line write and chat functions when at
different terminals, the use of mail and netfile to pass modules, code, ideas and so on amongst one another and so allowed the project to come together with considerable ease.

• I have learned how the work for a "big program" could be shared within a group to finally get to the aim of the entire group.... In this view each of us had to think of our pieces of code, but we had to never lose sight of ... others’ code. So we had to work in common; seeing each other really often so as to go in the same direction...

• Helping group members with their code taught me more than I could by myself.

• A clearer and more accurate plan in distributing work needed next time in order to minimise obstruction to the project’s progress.

• I felt that my group in this project has been very good to work with. We had regular group meetings so that every member was informed with what tasks have been completed and what needs to be done. The meetings also allowed the group to plan the program and solve problems together.

• I find that I am slightly more comfortable working in groups.... I relied less on the textbook for new information and tried things out instead.

• The last but not least of my goals was a challenge for me (the "group" goal). I am sure that in all the ways it brought me something special; in a human way as well as in a computer way. I learnt to not only think of me in my work but also of the people I was working with and for. I learnt how to learn more by sharing what they understood and I did not, and vice versa.

• I have learnt, for the first time, that co-operation and groupwork is just as important as being able to work independently.

Planning and Time Management

• Earlier on in the project I realised that our work to develop parts of the same program was a lot harder because we did not properly define an overall structure for the program and therefore could not properly predict how our program would fit in. When we got together and came up with a plan, coding became much easier and our separate pieces of code fitted together well.

• Putting the components of the project together is just as difficult as doing the individual bits.

• The only problem with our approach was that we did too much planning, leaving the coding for later. If we had a hypothetical computer that could make code out of our concepts we would have had a lot less trouble!

• We worked extremely well as a group, at least 50% of our work was done outside of our normal lab times (i.e. during breaks and weekends). All the members of the "Evil Mongers" [the name the group gave themselves] were very dedicated to the task. The general opinion of the group is that with more time we could have put together a very good system. Many of our ideas were turned aside due to lack of time.

• Time flies.... never, Never, NEVER, am I going to let myself experience the "last minute rush" again!!!!!!!
A student who wants to do well in the course should...

• study, be interested and spend a lot of time practising.

• be motivated, have an open mind, be interested in what they are doing and learn the basics quickly.

• practise, practise, practise!

• write lots of different code and play around with it a lot.

• practice.

• practice programming.

• listen to other people and PRACTICE! Practice makes perfect!

• Firstly understand the basics. You need to know how to walk before you can run!

• dedicate themselves to practice programming.

• do a lot of programming.

• program as much as possible - no use reading if you don’t practice.

Good Programmers Always...

• think before they start writing programs.

• fully understand the task at hand.

• write well-structured programs.

• think before they jump in.

• plan ahead.

• think before they code.

• read and understand other peoples’ programs, and practice, practice, practice.

• step back and think again when necessary.

• spend time to think through the problem, research, and practise programming.

• put in comments in order to help themselves to know what they are actually doing and to help others who are going to read.

• practice.

• know the basics.
Learning at University ...

- is really interesting.
- is interesting in the right environment.
- requires study!
- requires some more private individual effort.
- is different.
- is not easy, requires effort.
- is fun for some subjects (e.g. CS), but hell for something like .... [Sorry, we're not telling!]
- is much more fun than high school!! he he...
- is a great experience, but requires self-discipline because no-one is there to force you to do the work.
- is not easy.
- makes you more 'independent'. You really need to be clear of what is happening at the moment in order to go to the next step.
- needs perseverance.
2. Overview of Introductory Programming 1001/1901

This section tells you a little about the whole semester. Some of the things in this section are taken from formal documents specifying the class syllabus. Most students will not understand all the terms at the beginning of the semester. Even so, skim over it to see if you can work out roughly what it seems to mean and make sure you return to during the semester to see how it is coming to make sense. It had better make a lot of sense before the exam.

We begin with the overall objectives. These describe what will achieve by the end of the semester. Then we give an overview of how you will do this. Our presentation is top-down, starting with the things you will do over the weeks of the semester, then looking at the structure of each week. Finally, we show you the organisation chart for the course.

2.1 Course objectives - what you need to learn

These are the official objectives the department uses. The lecturer in charge of each course is required to report to the department on the progress of each course in terms of the these. So you can rest assured that there has to be some assessment of each aspect.

Preconditions: (What you need to know before you start)

Nil.

Postconditions: (What you will know at the end of the course)

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.

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;

In addition, when working in a group, students will be able to analyse a problem, design appropriate classes for solving it, document this design in terms of a Blue system model, combine parts written by individuals, and systematically test it.

To Pass this course, the student will perform these tasks at a minimal level as defined in problem assessment specifications and the examination instruction.

To earn higher grades, students will demonstrate the ability to do the above more elegantly or thoroughly, and to complete more challenging tasks.

### 2.2 Overview of assessment for the semester

Computer Science is great fun, but we also give you marks for doing it! This chapter outlines the assessment components so that you can develop a feel of how to maximise your results. You will get the opportunity to apply many of these same assessment criteria to your own work and those of your fellow students; this will be done before submission for assessment, and ensures that there won’t be any cries of "Oh, I didn’t know we had to do that!". We want you to get good marks. Its all part of the service...

The main structure of the semester is driven by the two major problems you will do. Problem 1 is a warm up for the major Problem 2. The table below lists the major deadlines associated with the two problems of the semester.

<table>
<thead>
<tr>
<th>Week</th>
<th>Assessment Task</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Problem 1</td>
<td>Group</td>
</tr>
<tr>
<td>1-4</td>
<td>Reflective diary, plan and progress data</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>Individual certification</td>
<td>10*</td>
</tr>
<tr>
<td>2</td>
<td>Individual certification</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Individual certification</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Individual certification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Problem 2</td>
<td></td>
</tr>
<tr>
<td>5-12</td>
<td>Stage 0 - Plan and progress data</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Stage 1 - Prototypes, Acceptance tests</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Stage 1 - Prototypes, Acceptance tests</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Stage 1 - Prototypes, Acceptance tests</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Stage 2 - Certification</td>
<td>10*</td>
</tr>
<tr>
<td>9</td>
<td>Stage 2 - Certification</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Stage 2 - Certification</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Stage 3 - Individual code</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Stage 4 - Demonstrations</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>Stage 5 - Reflective diaries</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Practical work totals</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Written Exam</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Grand total marks</td>
<td>200</td>
</tr>
</tbody>
</table>

† Note that Problem 1 gives no marks for the group planning and progress evaluation. This is because there will be many class changes in the early weeks. However, your tutor will be treating these in the same way
the problem 2 reflective diary, plan and progress data. This gives you a low risk way to learn what is required.

†† Throughout the semester, you need to keep your personal planning and reflection data so that you can write the report for Stage 5.

* indicated the ‘Certification’ tasks.

The Problem 1 Certification is required for progress to Problem 2. This means that if a student cannot get Problem 1 completed to a satisfactory standard by week 4, they cannot start Problem 2. They need to keep working on Problem 1.

The Problem 2 Certification is required for an individual to be entitled to the group marks. Essentially, Problem 2 Certification is earned when an individual has written a part of the group project to a satisfactory standard, as defined by the assessment criteria for Stage 2 of Problem 2. Only students who achieve this certification will be entitled to the credit the group marks for their group’s project.

2.3 Overview of topics in each class

The following list is an outline of the topics treated in seminars and tutorials. Note, however, that PBL is a highly responsive way of teaching. Some classes will need to make some modifications to this plan. For example, if your tute class wants more time spent on some aspect that people are finding difficult, there can be some shuffling.

A good thing to do on first reading through this list is to compare it with the headings in the Blue textbook and write on the chapter references against each week.

**Week 1**

**Seminar:** Welcome, course outline, variables, tracing, print
**Tutorial** Getting started, simple classes, code reading by making sense and guessing then using this to drive planning.

**Week 2**

**Seminar:** Routines and classes (tour of Account class, with tracing)
**Tutorial** Co-operative problem solving, listening skills, generalisation, code tracing, generalisation from traced code, control flow, especially loops.

**Week 3**

**Seminar:** Statements and lists (if, loop, basic list traversal)
**Tutorial** Self-assessment and concept inventories, criteria for evaluation; read code, write code, evaluate code, the Blue object model.

**Week 4**

**Seminar:** Communicating objects
**Tutorial** Control flow, tracing code and generalisation, especially loops; visual thinking, simplification (and its role in prototyping).
Week 5
Seminar: Prototypes, planning; code cliches (traversal, linear search)
Tutorial simplification in design; brainstorming revisited for starting Problem 2; heuristics for testing; self-assessment tasks for control flow aspects of Blue; designing good tests intuitively for small code segments; quiz.

Week 6
Seminar: Testing; more code cliches
Tutorial group skills, running a meeting; code quality, code review, code as communication with people and machines.

Week 7
Seminar: Object-oriented design
Tutorial code cliches quiz and review; self-assessment tasks on code cliches, guessing things that belong in libraries

Week 8
Seminar: Collection classes
Tutorial Code reviews focussing on correctness issues.

Week 9
Seminar: Program correctness (pre&post, invariants, boolean expressions)
Tutorial Code reviews focussing on style and design issues; object model self assessment.

Week 10
Seminar: Object model
Tutorial Design and planning final demonstration.

Week 11
Seminar: Rest of semester; common programming infelicities
Tutorial Code reviews on individual code, clarify how well it meets assessment criteria.

Week 12
Seminar: History, evaluation of OO languages, OO Design Case Study
Tutorial Software engineering themes and reflection on group operation.

Week 13
Seminar:
Reflection, feedback, the exam
Tutorial Concept inventories + self-assessment tasks + model examination.
2.4 How you should plan to work each week

The number of hours that you are expected to commit to a subject is determined at a higher level in the university hierarchy than our department. Many students report that while for many of their subjects the recommended 12 hours per week is sometimes not needed, this is not the case in Computer Science. In this course you should prepare yourself for this level of commitment throughout the semester. Every year a small number of computer science students find themselves in difficulty because they were able to convince themselves that the "12-hour rule" did not apply to them; ensure that you are in the category that makes regular progress in the subject and you will avoid these difficulties. As group work is an important feature in this course, you may need to exercise some vigilance in (subtly?) helping your co-workers to meet their commitment (this is no different from the "real world", and is a great leadership and management skill!). You will also be doing them (and yourself) a great favour in terms of performance in this course.

The twelve hours that it is expected that you will need to devote to the course may be broken down as follows:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Hours per Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 seminar session</td>
<td>1</td>
</tr>
<tr>
<td>1 tutorial session</td>
<td>2</td>
</tr>
<tr>
<td>1 workshop session</td>
<td>3</td>
</tr>
<tr>
<td>Extra lab time (programming)</td>
<td>2</td>
</tr>
<tr>
<td>Code design, Self-directed learning</td>
<td>4</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>12</strong></td>
</tr>
</tbody>
</table>

Each week in the semester, you will be required to submit a Reflective diary and plan. This will be discussed in the first week’s practical class. In you 3 hour lab each week, you will create a plan for the coming week’s work, for both class times and your private study time. Each week, you will bring to class the evidence of work you did in the preceding week. Example of this are:

- your written answers to self-assessment tasks, end of chapter tasks, extra tasks offered by your tutor;
- print outs of code you wrote for self-assessment tasks, and the like;
- print outs of code you wrote for tasks you devised yourself;
- print outs of code you wrote for the current problem;

2.5 People who run the course

An integrated education will usually be student-centred, active education. Teachers cannot integrate material for their students.

-- R. McClintock

2.5.1 Organisational Chart

From looking through the Departmental Handbook and other documents, you will know that academic departments have a clear structure. For example, there are academic (teaching) staff, administrative staff, casual tutoring staff, programmers, research associates, technicians, etc. In addition, the academic staff

   http://www.ilt.columbia.edu/academic/texts/mcclintock/pp/contents.html
consists of Professors, Associate Professors, Senior Lecturers, and so on. Of more immediate interest to you as a COMP-1001/1901 student, however, is the organisational structure represented in the figure below. This provides an orientation to the staff in terms of their relevance to you.

Figure 1. Organisational Structure from the COMP-1001/1901 Perspective

Note that the main part of this diagram has a "tree-like" structure, with you and other students at the top and the Head of Department at the bottom. The following paragraphs provide a basic outline of how these parts of the tree interact.

Fellow students

You will be involved in a group learning environment for this course, and will quickly learn to respect the opinions and skills of others around you. You will probably make some good friends amongst them. In the first four weeks of semester you will work with possibly all students in your workshop class in some way. This ensures that you feel confident in your workshop environment. After that, you will work in a fixed group of people on developing a program for the remainder of the semester.

Computer Science is a wide field, covering many areas of expertise. No matter how much world-wide acclaim some of our staff command in their field, however, they are almost certainly not the best people to ask to find an answer to problems you are experiencing with your modem software. Experience has shown that such problems are best solved using the expertise around you. Chances are another student has just recently solved the same problem with the same modem.

Once you get used to the idea of recognising the value in your fellow students you will find that your rate of learning soars. Its important not to restrict this to just computer-related topics; regard your fellow students as sources of valuable opinions and ideas. For example, a great way to make a start on a project is to enter a "brain-storming" session with other students.
**Tutors**

You will meet your tutor in weekly workshop sessions (these are three-hour sessions in a computer lab) and in tutorial sessions (which are two-hour sessions in a classroom). The workshop is heavily driven by the needs of students (you and your group members), and the tutor is present for one of the three hours to provide practical assistance and advice. This works very well once the groups have developed a sense of autonomy. However, at the beginning of the semester tutors will be present for much longer time periods in order to also offer assistance in settling into the workshop mode.

Tutors are a great source of feedback for your ideas, although their job is not to feed you with raw information. As many of our tutors are wise people, they have been given special training to avoid taking on the appearance of fonts of all wisdom. Think of them as wise, but without "Attitude"... Or "wise in disguise". No, they cannot even be taunted into providing you with direct answers to questions that they know you would benefit from investigating for yourself. Tutors are very careful not to deprive you of learning opportunities. Rather than attempting to radiate information for you to absorb, our tutors will seek to stimulate, provoke, and guide you in your learning. The approach is one of facilitation rather than indoctrination. However, as well as being wise, our tutors are caring people who will help you avoid pitfalls and lend a hand when you need it.

**Section Leaders**

You will meet your section leaders every week in the one-hour seminar session. They will be responsible for co-ordinating the tutorials, as well as marking some of the presentations and code submissions (some marking will also be done by your tutor). Section leaders will deal with broader issues relating to the course.

**Undergraduate Coordinator**

The undergraduate coordinator is Dr. Jeff Kingston, who has administrative responsibilities which extend to all undergraduate courses in the department.

**HoD**

The Head of Department, Associate Professor Bob Kummerfeld, is (as you would expect) the ultimate authority in matters relating to the running of the department.

**Computer Science Office**

You should identify the location of the Computer Science office. The departmental administrative staff ensure that many of the behind-the-scenes tasks get done. You should note, however, that the Computer Science office is not necessarily the first port of call when students have problems or questions. The help desk (see below) is intended to bear the brunt of most enquiries, although you will need to see the (always helpful!) office staff in cases of misadventure (when you should complete a Special Consideration form), or in the rare case where a tutor does not arrive at a tutorial due to unforeseen circumstances (in which case they will attempt to organise a replacement).

This is also the place you need to take Special Consideration forms and supporting documentation if you are ill or suffer misadventure.

**Help Desk**

The Help Desk is a dedicated band of senior students who will help you deal with problems such as password failures, programming problems, changing tutorials, etc.

The help desk has received unbridled praise from the many students that they have helped, in a variety of problem areas from administration to software bugs. You should consider them as your first stop if you are confronted by any sort of problem outside of scheduled classes. See back cover for time and place.
**Other Important People**

There are many other people in the department without whose continuing efforts the department would grind to a halt. These are people that you may not have much (or any) exposure to. We have a team of programmers who administrate the systems and networks you will be using, and work to ensure that installed software behaves as intended. There are workshop technicians who maintain the state of the equipment. If you discover a bad terminal, simply creating an Xterm from your Blue window (this will make sense after a while) and typing the command `report` will lead you through some simple questions that describe the hardware problem. Next time you see that terminal (or printer, etc.) it will be magically working; the technicians are the people who respond to your "report" command, and perform the magic. Finally there is a PBL co-ordinator who looks after staff development and support materials for the course.
3. Problem 1 - the Better-Bank-Builder

3.1 Aim of Problem 1

With all the words about PBL (Problem-Based Learning), you should be expecting to work on problems. Big, messy, interesting problems, where you get to decide how to creatively solve the problem while you learn lots about Programming and Computer Science.

For the first four weeks of semester, you will work on Problem 1. This is a warm-up or start-up problem. The main work of the semester will be in Problem 2 but before you can do that, you need to start to get the feel of PBL.

In the real PBL, you will work in a group on a single problem that the group will be aiming to solve. The way this will work is that the group will discuss the problem and its solution, under the watchful mentoring eye of your tutor. The group will then agree on how to split the work, allocating each person in the group a fair share of the work. If all goes well, each person will do their work on time and well. Then you will all put the parts together and viola! You will have a complete solution to the problem.

This is a wonderful thing when it works well. However, it is quite challenging to make sure that the work split is done well, that everyone delivers and that what they deliver works properly with everyone else’s pieces. This is a major Software Engineering challenge that you will tackle in Problem 2.

In Problem 1, your group will only do the first part of the job. You will each tackle different parts of the solution to a problem. You will develop these individually. You will not put them all together. So what, you may ask, is the group there for? Well, the group learning is an essential part of PBL. By working in a group, you help others in the group and learn in the process. You look at what other people in the group are doing and learn from reading their code: perhaps because it is better than yours; perhaps because they have learnt some things you have not come to yet; perhaps because they do things badly and as you try to work out what they are doing, you and they discuss the code and both learn how to write it better. Working with other people is also much more fun!

To tackle Problem 1, you really need to

- read the Problem 1 statement below;
- attend classes in Week 1 to help get started;
- start working on it, to develop your understanding of it (since you should not understand it completely at this stage - but you should be able to use your skill in finding parts you do understand as a foundation for getting going;
- read the marking scheme for Problem 1 so you know what we want you to be able to do by Week 4 of semester.
- make times to meet your group members in the Computer Science Department terminal rooms to work together on getting started (the hardest part of a problem).

Problem 1 is designed to familiarise you with all aspects of the operating environment. You explore some existing examples of Blue code, and see the effects of modifying that code. You can use these in several ways for your solution to problem 1:

i. by reading existing code, you can get a basic understanding of how it works, then modifying it as your first efforts at programming;

ii. example code will help you see how aspects of Blue work and then you will be able to use this for parts of your Problem 1 solution.

iii. some of the code can be used in association with your solution code;
iv. some will do similar tasks to what your code must do and you can mimic them for your own solution;

Make sure you start Problem 1 by finding the simpler first steps. The tutorial resources will help you identify these. For example, a good starting point is to explore the code we have given you. You should also check through the resources on the back of this book.

A good way for your group to operate is for each person to explore different resources, at least in the early stages.

### 3.2 Problem 1 description

Your Boss wants to show banks how they can operate better. Part of this grand vision requires that your team build a simulated bank that runs on a computer. This way, your Boss can set up the bank one way and simulate the way it would operate. Then it could be set up to operate another way. By comparing the way these two run, you should be able explore the effect of different ways of running the bank.

For example, we might have a bank which charges for each transaction and then pays a good interest rate on the money in people’s accounts. A different way to do things is to charge no transaction fee but to pay less interest. By simulating what happens with various account profiles (each with different amounts of cash in them and different numbers of transactions) you can see how each of these two approaches works out.

Your Boss will be played by your tutor. You will work in a group on this problem, with each person in the group working individually on different parts of the program needed to give the Boss what they want.

To help you get started, the Boss has already paid Jeff Kingston to write some banking classes which will be the foundation for the future work on the better-bank-builder. You will need to pick your tasks to fit in with the overall goal of the better-bank-builder system and to co-ordinate with the other members of your group, so that each person does different parts of the job. The Boss has the provided the following list of things that you might do.

- change accounts so that they charge a transaction fee;
- make sure that you only charge that transaction fee is there is enough money in the account to cover it;
- set up accounts so that those which have lots of money in them do not get charge the transaction fee;
- ensure that customers get interest on their accounts: this can operate on a daily, monthly or annual basis as you like;
- ensure that when an account is created, its interest payment type is specified as one of daily, monthly or annual and then the interest in added accordingly;
- add very simple single-user *Teller* class which works like a teller interface, welcoming each the user, asking if they would like to make deposits, withdrawals... and then doing them for the person and reporting the final balance after the transaction;
- add a customer class which keeps track of all the usual things that a bank would need to know about a customer if the better-bank-builder system is to work well;
- upgrade the *Teller* so that it asks for the user’s name and then operates on their account;
- upgrade the *Teller* so that on withdrawals, it prints the actual notes and coins paid;
- ensure that the bank can have lots of customers;
- and that customers can have multiple accounts;
• and a customer can transfer money between their own accounts;
• and enable accounts to have multiple signatories;
• provide an automatic payment option where a customer can specify a list of bills and dates that they should be paid;
• provide a daily transaction record for an account, so that the *Teller* can provide a list of all withdrawals and deposits;
• a monthly transaction record;
• write code which enables a simulated ATM to be loaded with a large amount of cash in various denominations, which it can pay out to a customer (eg for $100, 5 $20s, or 10 $10s, or 2 $50s);
• implement the ATM so that it allows a fixed, small number of attempts to enter a pin, after which it gives a message stating that there have been too many attempts and that the card has been disabled for 24 hours;
• implement the disabling of the credit card till mid-night;
• apply a maximum daily withdrawal limit;
• and the Boss may also be quite happy for you to build other code-tools which will be useful for the better-bank-builder system. You will need to consult your Boss in class about this.

Your own Problem 1 work will involve some of these, while your group members’ will involve others.
3.3 Detailed assessment for Problem 1

It is important that you understand how you are assessed. You should work through this marking scheme before you hand in your Problem 1 folder. We want you will develop skill in assessing your own work. You are required to self-assess your work. Take the copy of this marking scheme at the end of the notes, mark it with your self-assessment by ticking boxes for criteria you believe your work achieves. Attach it to your folder.

Note that no fractional marks are awarded

Marked by you tutor, in 3-hour workshop, by week 4

On the marksheet you submit, annotate the marksheet to indicate where the marker will find aspects with Location.

Criteria for a pass and certification in Problem 1 (5 or 6 out of 10)

You must achieve a Pass before you can begin Problem 2.

Required criteria for the grade Pass

The submission must have all of the following elements.

- It is in a folder with the code you have written, the test report and statements, all firmly secured and well presented.
- 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 completed self-assessment for the problem must be attached to your folder.
- Your code should be demonstrably able to do the job it is supposed to do: it must work.
- The code to be assessed should be no more than 4 pages long. Location:
- Your submission must include one whole class. Location:
- Your code must make use of at least one loop. Location:
- It must make use of at least one if-statement. Location:
- It must make use of at least one LList. Location:
- 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 1 page.
- 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 criteria for the grade Pass

At least three (3) of the following must hold:

- 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, postconditions and class invariants. **Location:**

- 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 (7 out of 10)**

All criteria for Pass as well as all * items and at least 3 more:

- * Must be handed in on time: no later than week 4’s 3-hr workshop.
- * Testing is convincing (within 1 page 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.
- All the Blue control structures should be appropriate for the task.
- There should be no 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.
- Your code includes references to some sources you used (for example, if it is based on information from an accounting book, it should give the title, author, publisher, year, pages you used - information from a bank could be the details of the document - if you used some of our Blue resources, state the page/url and how it was used)
- Good attempt at correctness support: preconditions, postconditions and class invariants. **Location:**

**Criteria for a Distinction ++ (8 or more out of 10)**

All Pass and Credit criteria plus some of following (for 9+, all starred items, for 10 all items):

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

Note that if you are in COMP1001 and would like to be considered for COMP1901, you can demonstrate your ability on Problem 1. If you can achieve a high grade and on work submitted by your 3-hour workshop in week 3 (repeat three), your tutor will discuss transferring you to COMP1901. (Note: even if you have no aspirations for COMP1901, you are most welcome to complete Problem 1 early. Your tutor will be delighted to be able to mark it ahead of the rush in week 4.)
4. Problem 2 - simulations

Welcome to Problem 2, the main task for the semester. To get this far, you must have achieved certification on Problem 1.

4.1 Problem 2 - the problem statements

Ambulance dispatch strategy exploration

Imagine you are in charge of dispatching all the ambulances in a city. There are many ambulance stations. Some have just one ambulance, some have two and others have several. When a call comes from a patient, you need to decide which ambulance to dispatch. You would like to be able to explore several possible strategies for doing this. To see the effect of a particular strategy, you would want to see the simulated, changing locations of ambulances as time elapses. As the dispatcher, you would want to be able to know how things are going, as ambulances move from their stations, pick up patients, take them to hospitals, return to their stations (possibly taking a call en route).

One, very simple strategy might be to send the ambulance which is closest to the patient, as the crow flies. A slightly more complex strategy would send any ambulance within x kilometres (or y minutes) of the patient, but avoiding taking the last ambulance at any one station. There are many more strategies you might devise. You can also include varying levels of refinement: model the degree of severity of the person’s problem and hence the urgency of the call; model the long term load on ambulance personnel so you can avoid repeatedly calling the same people..... You could also maintain running statistics on how things are going: average time for ambulances to get to their patients, and to get patients to hospital, the number of patients who died because the ambulance was too slow.... You may want to make simplifying assumptions: for example, you might assume that all streets form a perfect grid so you can calculate the time an ambulance takes to get to a person and then to a hospital. We will provide some example data sets - to see these, use ls "cs1/comp1001/life*" (for more on ls etc, see the unix information in this book). We will post on the web, details of systems that perform well on these.

Your final presentation for this project would show how a dispatcher would use the system to see the effect of some strategies.

Fire management simulation

Imagine that you are in charge of the education programme at a national park. You are building a system which enables visitors to explore approaches to fire management. The system should enable them to see the various areas of the park, some densely forested, some with windswept areas with only low, sparse vegetation, some areas with water flowing through..... You may also model paths in some areas, with many people travelling through as well as other areas that have very few people entering. At times, fires occur. Depending upon the conditions at that simulated time, these fires may spread quickly, or they may burn themselves out, perhaps meeting a wet area or an area without fuel for the fire.

Your system could allow users to see the effects of different fire management strategies. For example, stopping people from entering an area should reduce the risk of fires started by smokers and people who light fires. Another strategy might involve controlled burn-off at suitable times. There are many factors to simulate. For example, you could simulate the way that large trees and other dense foliage drop dead leaves and branches, so creating fuel. You might decide to simulate the effects of fires on wild-life. For example, koalas move slowly and may be caught in fires. Even if they do escape the fire, they may die if too much of their food supply is destroyed.

Your final presentation for this project would show how a user might see the effects of some fire management strategies. You should point out the educational goals of your system - what you would expect people to learn from using it.
Crowd management planner for Olympics venues

Imagine the minutes after the completion of the finals for the women’s swimming events at the Sydney Olympics: the whole audience will want to leave the venue. Your job is to assist planners in exploring options for managing this. You will write a system which simulates people leaving olympic events. Then planners can see the effect of changes. For example, they can create additional exit doors, block off existing doors, enlarge doors... There can also be multiple routes to each form of transport. For example, there are routes that involve stairs, others that do not, different paths to opposite ends of the bus and train areas.... The planner will want to see how different strategies seem to affect the time it takes people to leave an event. (They will be interested in several measures: the mean, median and mode time, fastest, slowest, etc.)

Each person will try to make their way to an exit (usually the nearest one, though a person seeing some other exits that seem less crowded might try those). Then they will make their way to their transport home. A few will walk to their accommodation at the olympic village. Most will walk towards the railway station and the bus stops. Some walk to cars or taxis. Some people with handicaps will need to take routes that avoid stairs and some may move slowly. Some people may decide to delay their departure, so they can chat, visit the coffee shop, the toilets, or wait to see another glimpse of the swimmers. You would need to run the system with various combinations of population split: for example you might start assuming that everyone walks at the same speed, is catching a train and leaves the venue by a single door.

Your final presentation for this project would show how a planner might run your system with one combination of exit structures (doors, paths....) and types of people. You will also present another scenario, with a different combination of exit structures and people. Demos will start with everyone seated at the end of the event, simulate their movement towards an exit, then out and off to transport.

Investment strategy exploration

Imagine that you have just won a prize of $1,000,000 (for an outstanding program you wrote). You need a system that will help you explore the impact of different ways of investing it. This will allow you to allocate the money in various ways, then see how things might work out in the long term, say 25 years. You want the simulation to show the possible effects of typical rises and falls in stocks, some major changes at random times that simulate major events such as changes in interest rates. You are interested to explore how various scenarios might work out in terms of the annual investment income, final capital, estimated risk... This project is only suitable for groups which include some students who are keen to do some research on the performance of various investments.

You might use some of your money to buy various shares, property, or keep some in cash. You might want to explore other possibilities too, like a regular investment in Lotto, high-tech start-ups.... Note that you may want to simulate the effect of tax on each investment. You will want to see how various strategies work out. You should ensure that your system can show the effect of the investment possibilities indicated in the supplied files for this problem. (To see these, use ls ~/cs1/comp1001/investment* -- for more on ls etc, see the unix information in this book.)

Your final presentation for this project will show what you would see about the performance of various strategies at various point in time. So, for example, you might show all money on shares and what happens to that investment in each month. Then you might try half in shares and half in Sydney commercial property. You should demonstrate using some of our supplied data on stock and property performance as well as some of your own (and you will explain why you chose it and how it is more interesting than our data.)

Phone plans simulator

Imagine that you and a friend are about to acquire a mobile phone. You see the bewildering array of possibilities with various plans. Some offer cheap entry and may be cheap if you don’t use the phone much. Others are expensive at entry but seem to be cheaper for heavy use. Also, some have good deals for late night use. You will build a system which enables you explore the ongoing costs for various plans.
Note that you and your friend have different expected profiles of use so the system must be flexible enough to let you explore a range of possible usage patterns that may change over time.

You would want to take account of the fact that it is hard to predict just how much you will use a phone. So, you should show the long term cost over 2 years, with various levels of use, perhaps a burst of calls at times of birthdays etc of your family and friends, at festive times etc. You should also show the effect of making heavier use, or lighter use than you initially predict. (To see some scenarios, use `ls ~cs1/comp1001/phone*` -- for more on `ls` etc, see the unix information in this book.)

Your final presentation for this project would show each monthly bill over a couple of years, with the expected cost each month, for various strategies and various usage patterns.
4.2 Problem 2 assessment criteria

4.2.1 Problem 2 - Stage 0 - Planning and progress data

This is a critical part of the work you will do this semester. It assists the tutor in tracking how the group is operating. Importantly, it helps group management.

Criteria for a pass (10-12 out for 20)

Required criteria for the grade Pass

☐ In at least 5 weeks, there is a serious attempt at a group plan submitted via the on-line system.

Criteria for a Credit (12-14 out of 20)

The Pass criteria plus one mark for each 2 of:

☐ Solid and consistent progress.
☐ Effective use of programming-by-contract.
☐ Able to demonstrate a working (even if incomplete) version by week 9.
☐ Regular, thorough evaluation and testing of each person’s code: one member of group tests work by another member.

Criteria for a Distinction ++ (15+ out of 20)

All the above criteria plus 1 for each of:

☐ Creativity in defining interesting things to explore and ways to learn (eg. sharing the exploration of new learning areas).
☐ Excellent management of testing each class (this should mean that integrating the code was smooth).
☐ Overall project planning is robust, so that if any one member of the group were to suddenly stop working, the project should proceed successfully.
☐ Regular code reviews, with each group member’s code being reviewed by whole group at least once.
☐ Built minimal but working version early on, then enhanced this as semester proceeded, having a working version of some sort for much of the semester.
4.2.2 Problem 2 - Stage 1 - Prototypes plus acceptance tests

To be marked by your tutor, in the 3-hour workshop no later than the week 7.

Criteria for a Pass on Stage 1 of Problem 2 (5 or 6 out of 10)

Required criteria for the grade Pass: The submission must have all of the following elements.

- A folder containing all required parts of the work firmly secured and well presented.
- It must begin with a signed statement that every member of your group has read the final form of the whole submission and this must be signed by each member of the group.
- The functional prototype part of the report will have the project diagram for the code written and up to 1 page of example input/output.
- The structural prototype part of the report will have the project diagram and the class interface view of the major classes.
- The Acceptance Tests part of the report will state the tests that you would expect to present at the final demonstration of the semester and it will be no more than 1 page long.
- The functional prototype demonstration provides one scenario of the system operation: the marker will observe a running Blue program that shows what the complete system would do in one well-chosen case.
- The functional prototype gives a good indication of what the final system will do.
- The functional prototype code includes a comment explaining how the final program will solve your groups’ problem;
- The structural prototype design has at least the main classes needed to make the simulation actually do its job, simulating the activity required for your understanding of your chosen problem.
- The structural prototype design includes comments stating which group members are responsible for each class.

Criteria for a Credit (7 out of 10)

All the Pass criteria plus all starred (*) items and at least 3 other items below.

- * The complete submission must be handed in on time.
- The functional prototype gives a clear idea of what the group intends to do.
- The functional prototype makes it quite clear what the group intends the full program will do and it is quite clear that this will address the problem chosen.
- The structural prototype design looks sound, clear and simple.
- The structural prototype design gives the most important classes in enough detail for the marker to see the instance variables and the interface routines and this is mainly well done.
- The acceptance tests are well chosen and convincing.

Criteria for a Distinction ++ (8 or more out of 10)

All criteria for Credit plus 1 of the following for 8, 3 for 9, all for 10.

- The structural prototype design is elegant.
- The structural prototype design is at a level that the future development is a simply matter of upgrading current classes.
- The detailed design of the classes in the structural prototype has good choice of instance variables, the interface and correctness support to match.
- The acceptance tests are simple, elegant, clear, minimal.
4.2.3 Problem 2 - Stage 2 - Code certification

To be marked by your tutor, in the 3-hour workshop no later than the week 9.

Criteria for a pass and certification on Stage 2 of Problem 2 (5 or 6 out of 10)

You must earn a Pass if you are to be entitled to the group marks.

Required criteria for the grade Pass

The submission must have all of the following elements.

☐ It is in a folder with the code you have written, the test report and statements, all firmly secured and well presented.

☐ It must begin with a signed statement that the code you have submitted was entirely written by you (and if you needed a significant amount of help for any other aspect of the code, you should explain that near this signed statement).

☐ It must have a statement that the code is part of the group’s project and this must be signed by you and all members of your group (where this can include, for example, a class that tests out another class that will contribute to the final project - such a testing class may not actually be part of the final project but does contribute to the group’s work; another acceptable example is a class which explores one way to do part of the problem, while someone else writes another class which explores another way - only one will be part of the final project but so long as the group agrees that both approaches should be explored, both are contributions to the group project and this situation should be clear from the group planning sheets; yet another example is where the group decides that one part of the program is critical to the success of the project and therefore asks two group members to work on independent versions - here too, only one will be part of the completed project but both are part of the group’s agreed work on the project).

☐ The code to be assessed must be clearly indicated, perhaps with a cover sheet stating where the marker should look and if this is a particular routine, it should be clearly marked with highlighter along the edge of the page.

☐ The code to be assessed should be no more than 5 pages long.

☐ Your submission must include two whole classes.

☐ Your code must make use of several loops, including at least one nested loop, several case and if-statements: a good testing class, which tests out another class, can easily need these.

☐ It must make use of at least two collection classes.

☐ You must submit a Test report which list the tests you did to convince yourself that it works correctly: maximum length is 1 page.

☐ In your 3-hr lab you must demonstrate that your code works by doing these tests (maximum time for this demo is 5 minutes).

Criteria for a Credit (7 out of 10)

All criteria for a Pass plus * items and 2 more of the ☐ items following:

☐ * The complete submission must be handed in on time.

☐ * There is an attempt to write preconditions, postconditions and class invariants.

☐ Good style and documentation - meaning

  • Layout must be consistent and clear, with indentation showing code structure.

  • 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.)

  • Each class should have a comment stating where you found the ideas and code that helped you write that class (eg the text, reference, on-line examples, other members of your group,....).
• Each routine should have comments explaining what it does in terms of its parameters and return values.
• Helpful comments explain each of the major instance variables.
• Names for classes, routines, parameters and variables should be helpful.
• Good use of const identifiers to avoid magic numbers and to ensure the code can readily be modified to do slightly different tasks.
• Good use of enumeration classes as appropriate, to make clearer values for data.
• There should be helpful comments through the code as needed.
• The spelling is good and overall English expression in comments is pleasant to read.
• Presentation of the code means the reader can understand it with minimal effort - it should be pleasant to read, with self-documenting code style and judicious use of comments.

☐ Good design - meaning
  • All the Blue control structures should be appropriate for the task.
  • There should not be tedious code. (If you need to write a swathe of print statements, make sure these are not a significant part of the code you submit for assessment.)
  • There should not be duplicate or near-duplicate code. (So if you could write a single routine which could do the task(s) of several repeated pieces of code, you should write that routine, reducing the overall length of your program).
  • Routines are local as appropriate, none available at interface unless necessary.

☐ Testing is convincing (within 1 page limit), stating: the purpose of the test; the input for the test; the expected output or behaviour; observed behaviour; presentation is clear.

☐ Checks on all user input to ensure that it is ‘reasonable’ and then the code should handle unreasonable data gracefully, perhaps inviting the user to enter the data again.

Criteria for a Distinction ++ (8 or more out of 10)
All Credit criteria plus * items, and at least 1 more for 9, all for 10. of the following:
☐ * Code does something interesting and challenging.
☐ * Good attempt at correctness support: preconditions, postconditions and class invariants.
☐ Code should be clear and simple.
☐ Each class should be well designed, with good choice of routines and instance variables.
☐ Test page is a delight to read, clear presentation, spelling correct, within space constraints, each tests assess a different aspect.
4.2.4 Problem 2 - Stage 3 - Individual Code

To be marked from the report and a demonstration in the 3-hour workshop no later than the week 12.

Criteria for a pass in on Stage 3 of Problem 2 (10-12 out of 20)

Required criteria for the grade Pass
The submission must have all of the following elements.

☐ It is in a folder with the code you have written, the test report and statements, all firmly secured and well presented.

☐ 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).

☐ It must have a statement that the code is part of the group’s project and this must be signed by you and all members of your group.

☐ The code to be assessed should be no more than 6 pages long, and you must indicate which parts, if any, were submitted for Stage 2.

☐ It must perform both input and output on files.

☐ You must submit a Test report which list the tests you did to convince yourself that it works correctly: maximum length is 1 page.

☐ In your 3-hr lab you must demonstrate that your code works by doing these tests (maximum time for this demo is 5 minutes).

Criteria for a Credit (13-14 out of 20)
All criteria for a Pass plus * items and at least 2 of the following:

☐ * Must be handed in on time.

☐ * Good attempt at correctness support: preconditions, postconditions and class invariants.

☐ Good style and documentation (as described for Stage 2)

☐ Good design (as described for Stage 2)

☐ Testing is convincing and clear (within 1 page limit), stating: the purpose of each test; its input; expected output or behaviour; observed behaviour.

☐ Checks on all user input well done.

Criteria for a Distinction ++ (15 or more out of 20)
All criteria for a Credit plus * items and 1 mark extra for others:

☐ * The amount and overall quality of the code written goes beyond meeting the minimal requirements listed above.

☐ * Code should be clear and simple.

☐ 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.

☐ Excellent correctness support, thorough and well formulated, and no instance variables used for interface (pre/postconditions).

☐ Delightful presentation.
4.2.5 Problem 2 - Stage 4 - Group demonstration and poster

To be marked by your section leader in the 3-hour workshop in week 13.

Note that the time limit for the demonstration is **10 minutes**. Any group that goes beyond this will be stopped at 12 minutes, regardless of what has or has not been demonstrated.

**Criteria for a pass on Stage 4 of Problem 2 (10-12 out of 20)**

**Required criteria for the grade Pass**

The submission must have **all** of the following elements.

- The group begins the demonstration by stating what the project is intended to do.
- And the demonstration indicates that it does this.
- Each member of the group participates in the demonstration in some way.
- Demonstration is pleasant to attend.
- The group explains how the system developed actually solves the problem and the demonstration indicates that it does achieve a significant part of that role.
- The poster clearly states the goals, how the system addressed the problem, the strengths of the system, shows examples of the system in operation, includes a list of the people in the group.

**Criteria for a Credit (13-15 out of 20)**

All criteria for a Pass plus * item plus 1 for each extra:

- * Demonstration within 10 minute limit.
- The demonstration shows a convincing range of the system operation.
- Presentation is well structured and easy to follow and the important points are clear.

**Criteria for a Distinction ++ (16 or more out of 20)**

All criteria for a Credit plus * item and 1 mark for each extra:

- * The demonstration appears well planned and rehearsed.
- Presentation demonstrates the system convincingly, showing how it is used for to address the central problem.
- System as demonstrated has some interesting aspects that go well beyond the basic task.
- Presentation is entertaining.
- Poster is clear, elegant and visually appealing.
4.2.6 Stage 5 - Reflective diary

Criteria for a pass on the reflective diaries (5-6 out for 10)

Required criteria for the grade Pass
For each week of Problem 2 (weeks 5-12):
- a summary of the work done in that week;
- a reflective assessment of that week’s work, how well, in hindsight, you consider that you worked that week: perhaps you wasted time at the terminal because you had not read enough materials; perhaps you wasted time because you tried to do too much at once, rather than breaking the task down and solving smaller parts; perhaps you were hampered by problems interacting with other group members; perhaps you did not work hard enough...

Criteria for a Credit (7 out of 10)
All criteria for a Pass plus * item and one other:
- * Reflective diary submitted on time
- Attachment of supporting documentation for claims about work done (eg attach personal plans for the semester, copies of exploratory code written)
- Diary shows insight into successes achieved as well as reasons for problems and how to address them.

Criteria for a Distinction ++ (8+ out of 10)
All criteria for a Credit plus one for each of the following:
- Diary shows how you assessed your progress on the project and on learning the course goals;
- Diary shows how testing strategies did or could have ensured smooth progress of the project
- Diary shows insight into how demonstrable achievements for the project were or could have been planned for each week.
5. Eclectic set of resources to support tutorial work++

This section has resources that will be used in tutorials, private study time, and at the computer. Some support the development of generic skills for problem solving. Even these will involve lots of programming activity since that is the most logical starting point for building those skills within this course.

5.1 Problem solving skills: individual planning

In Problem-based learning (PBL), you follow a sequence of steps in planning your learning. We provide an on-line system to assist with planning. Your tutor will want to review your plans regularly. You will need to review them at the end of the problem, when you reflect on the whole process involved in solving each problem. You should be able to develop your problem solving skills if, at the end of the problem, you reflect on the way you went about various aspects of the problem solving. The you can identify the things you did well and those where you can see how to do better in future.

Here are the steps involved in your planning.

i. **Problem statement.** State the problem as you best understand it now. Since PBL involves loosely structured problems, you should expect to refine your understanding of the problem as you progress. This is typical of problem solving in general: when you first start a problem, you do not know enough about the details of it and solution methods to be able to really appreciate what is involved.

ii. **Current subgoals.** List the subgoals you need to achieve to solve the problem as best you understand it at present. This includes the things you believe you need to know or do to solve the problem. For example, you might decide that the first thing you need to do is to learn some of the very basic aspects of Blue. We might call this Subgoal 1.

iii. **How you will know when you have succeeded.** This is really a tightening up of the last stage. It is very easy to decide on a sloppy goal like 'learn very basic aspects of Blue'. But if you leave it at that, how will you decide when you have achieved that goal. How will your tutor know what you mean by the goal? Also, when you work in a group, how will your team members know? It is important to define some demonstrable achievement you will use to judge success (or otherwise).

For example, for Subgoal 1, you might decide that a good demonstration would be to write a first Blue program that does something very simple, but of your own choice (perhaps print 'Hello world' in old computer science tradition, or read in two numbers and print their sum, ...). It is easy to decide if you have actually done that and it is a good demonstration that you have mastered some of the really basic aspects of Blue.

As a problem solver, one of the most important things you need to learn is how to set your own learning tasks. Different people, with different preferences, knowledge and skills will find different tasks suit them.

iv. **What you already know**. This means that you think about the things you already know that are relevant to the problem. It might include identifying the most important aspects of the problem, the things you already know and plan to use in solving it. If we think about Subgoal 1 above, you might note that you have used computers for word-processing so you have keyboard skills and confidence in using computers.

v. **Steps to take, by when**. This is the list of things you plan to try out to achieve each subgoal. For example, the Subgoal 1 of learning basic aspects of Blue requires you to do several things. For example, you might decide to do many things: explore the text book and see what seems helpful there; explore example Blue programs we provide; just play in the Blue environment; plan to work in the prac session with a friend so you tackle the task together. You might use any combination of these or other steps.

It is critical that you also put the time limit on each of these. It is very easy to wander aimlessly
around one resource and get nowhere. Part of your skill as a problem solver is to decide how much time you should put into one exploration. If that time comes up and you have not made progress, you need to stand back and rethink. Perhaps this approach is not getting you anywhere. Perhaps you need to stop now and see a tutor at the Help Desk. On the other hand, you might decide that a little more time should lead to real results. That is fine so long as you are disciplined and really do spend only a little more time. Computer Science is notorious for being a perfect way to waste hours getting nowhere. You need to know when to quit a task and to try something else or seek help.

vi. *Use of 3-hr group time.* The three hour lab period is a special time in the week. It gives you access to fellow students plus computers to work on collaboratively plus your tutor, who is dedicated to facilitating your learning. You need to decide which of the tasks on your list should above can be best done in that time.

vii. *Use of 6-hr private study time.* Just as the lab time offers special learning opportunities, there are many things you can do alone - in fact many of you will prefer to read books and spend much of you programming time alone. Students who work steadily, with well planned 6 hours of work each week tend to get through computing courses painlessly, without last minute panics and with little extra swatting for the exam. (The corollary is left as an exercise for the reader.)
### Planning sheet for individual work

<table>
<thead>
<tr>
<th>Description</th>
<th>Planning for the coming week</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Problem statement - current understanding of overall goal(s)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2. Current subgoals</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>What to do ...</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Things to learn, do, organise ... in the coming week in order to make progress on the main goal(s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3....</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3. How you will know you have succeeded (What to do)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For each subgoal in part 2, the way to conclude you have achieved that subgoal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3...</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4. What you already know (part of how to do the problem)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texts, class notes, peers, Blue, example code</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>5. Steps to take, by when (part of how to do the problem)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actions to do to achieve the subgoals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>6. Use of 3-hr group time</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List from point 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>7. Use of 6-hr private study time</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List from point 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Week 1 outcomes**

This is a summary of things you should have done or learnt by the end of the first week of the course. It is fine if some of these run into the second week but no further.

You are doing well if you can

- name at least one other student in the course, preferably more than one
- state your tutor
- state the login-id for your tutor
- be able to find the goals of the course and state how well these match your initial expectations
- send an email message to one fellow student, to yourself and to your tutor
- read email - there should be some for you
- get to a given url (universal resource locator - eg http://www.ug.cs.usyd.edu.au/~cs1)
- change your password
- know what is in the workbook and where
- start Blue
- use Blue to get the Account class loaded
- and then create accounts with heaps of money, deposit more, get the balance, withdraw some, see the balance again
- know how to look at the code of a class and have some idea of roughly what it is doing
- and its interface
- have chosen three small tasks to start working on in Problem 1
- be able to explain why this list is broken into the four sections.
5.2 Group problem solving : creativity

The goals of this activity are to:

- do group problem solving
- benefit from the creativity boost of a brainstorming session
- use strategies for tackling new problems where the goal is unclear
- use strategies for working out how to do things in Blue by using the printed resources
- learn a little about Blue by reading it
- and working with other peers in trying to make sense of an unknown program
- learn what you do not know but think you are ready to follow up from the Blue resources
- appreciate that you can understand some parts of a program in blue if you can read English and if the programmer has done their job well
- help you get started on Problem 1 both by working in a group to identify things to do to get going on it and to tackle reading some Blue code from a cold start

You should do two brainstorming tasks:

1. form a group and together brainstorm about things to do in the next week as a start for Problem 1. (Note that if you are unsure what this Problem 1 is all about, the brainstorming should include strategies for solving that problem - and as a helpful hint, we can tell you that it requires you to write some significant piece of Blue code.)

2. form a group and together study the code below, taking turns in reading it. The group should discuss what they think the code means, each person annotating their own copy to show the parts they do not have any idea about, also marking things they think they know but are unsure with the whole group

About brainstorming to enhance creativity

Brainstorming is a strategy for a group to do creative and lateral thinking. There are two essential elements:

- each person should be non-judgemental and non-critical of any suggestions
- there must be lost of relaxed activity for greatest effectiveness

Note that normally people are briefed on the problem well ahead of the brainstorming session so they can think about it - in this case we believe that students have been thinking about it. Students should take the following steps:

- select a group leader who will excite the group and play the role of encouraging comments from all members and will encourage people to build upon each other’s ideas, including really outrageous ones.
- spend 10 minutes brainstorming with all suggestions being recorded - students should aim to fill several sheets and should be encouraged to write goals of the process as well as actual things to do
- spend 10 minutes organising the ideas into related groups and writing a coherent ‘report’ of the session

Then you will be asked to share your group’s results with the whole class.

Note that brainstorming has value when lateral and creative thinking is needed. It works well only for small problems.
class Account is

== Author: Jeffrey H. Kingston
== Version: 1.0
== Date: 9 March 1999
== Short: a bank account
==
== Objects of this class define bank accounts with an owner and a current
== balance. Owner, balance, deposit, and withdrawal operations are provided.
==

uses

internal

var

  _owner: String
  _balance: Real

interface

  creation(owner: String) is
    == create account with given owner
    pre
      owner <> nil
    do
      _owner := owner
      _balance := 0.0
    post
      owner <> nil and balance = 0.0
    end creation

routines

  owner -> (res: String) is
    == account owner
    pre
      true
    do
      res := _owner
    end owner

  balance -> (res: Real) is
    == current balance
    pre
      true
    do
      res := _balance
    end balance
deposit (amount: Real) is
  == deposit amount dollars
  pre
  amount > 0
  do
    _balance := _balance + amount
  post
    balance = old balance + amount
end deposit

withdraw (amount: Real) is
  == withdraw amount dollars
  pre
  amount > 0 and balance >= amount
  do
    _balance := _balance - amount
  post
    balance = old balance - amount
end withdraw

invariant

  _owner <> nil and _balance >= 0

end class
5.3 Problem solving skills: problem solving style, resources, listening

This exercise has three purposes: to increase your awareness of how you go about solving analytical problems, so that you can do so more efficiently; to improve your ability to use the printed Blue resources; and to help you develop skills as good listener. All will contribute to your ability to work on problems this semester and later.

Form a pair or triple. Nominate one person as the problem solver and the other(s) as the listener(s). The problem-solver sets to work on the problem, continuously talking about what "s/he" is thinking. The listeners encourage this and monitor progress -- if they don’t understand what the problem solver is saying, they say so and force the problem solver to explain, thereby encouraging accurate thinking.

Take turns being the problem solver. Do as many problems as you can. Referring to class materials, such as the Blue Language Specification, is an essential part of these tasks.

i. This program fragment does not do what its comment says it does. What is wrong, and how would you fix it?

```plaintext
-- deduct transaction fee of $2 from balance if possible
if balance >= 2.0 then
   print("transaction fee of $2 deducted \n")
end if
```

ii. This program fragment does not do what its comment says it does. What is wrong, and how would you fix it?

```plaintext
-- add 5% interest to balance
-- (or 2% if balance is less than $200)
if balance < 200.0 then
   balance := 1.02 * balance
end if
balance := 1.05 * balance
```

iii. Here is the ‘deposit’ routine:

```plaintext
deposit (x: Real) is
   == deposit x dollars
   pre
   x > 0
do
   balance := balance + x
end deposit
```

Change it so that it gives a free $200 bonus if the owner’s name begins with J.

iv. Actually, in the account class the ‘deposit’ routine has postcondition

```plaintext
post
   balance = old balance + x
```

What does this seem to mean? Is that what it really does mean?

v. The following program fragment has a syntax error, that is, it is not well-formed. What is the problem? Use the syntax diagrams beginning on page 139 of the workbook for syntax checking.

```plaintext
if balance = 0.0 then
   print("account is empty \n")
else
   print("account contains $ ", balance, " \n ")
```

vi. The following program fragment is supposed to work out the sum of the first ‘n’ integers, but it too has a syntax error. Find it.

```plaintext
i := 1
```
sum := 0
loop
  exit when i > n
  sum := sum + i
  i := i + 1
end loop

Once you have finished this task, fill in the following questions on problem solving style. In doing this exercise, you saw another student solving problems. Now sit alone to reflect on your own approach to problem solving by completing the following questions about yourself. Then answer them for your partner. After you have both completed the questions, compare answers and then reflect on how you see your approach to problem solving.

Your initial emphasis is on:

---

Accuracy | Speed
---|---

You

work quickly, sometimes assuming previous ideas are alright, without checking

Check and double check, taking a long time to complete the problem

You

Take a wholistic view, solving the whole problem at once, rather than thinking in terms of subproblems

Take an overall view but break problem into subparts

Quickly identify parts of the problem and attack these one at a time

When stuck, you

Go back and check things carefully

Make a good guess and go on

You do all your thinking

in your head, spending a lot of time silently thinking

with paper and pencils, writing things down

Now do these questions for yourself as a listener and for your partner as a listener.

How much was said: I found the listener

passive - hardly knew they were there

about what I wanted

continually interrupted so I could hardly talk
What was said: I found the listener very threatening about what I wanted excessively supportive

I would characterise the listener as:

too silent neutral sounding board coached leader

The listener’s emphasis was on: I would characterise the listener as:

listening to me helping me to verbalise helping me to solve the problem solving the problem for me

Note that different problem solvers have different preferences in solving problems and different needs from a listener.

Now, reflect on your preferences as a problem solver. How do you prefer to tackle problem solving? (eg. Did you check back on things? Do you externalise your thinking with written or drawn notes?)

Now, reflect on yourself as a listener? Did you find this role easy? Did it help you learn about the processes in problem solving? Were you surprised by your partner’s problem solving strategies? How did they deal with getting stuck?
5.4 Problem solving skills: tracing and generalisation

The following piece of code † is the focus of this activity. You should work in a group to read the code and try tracing it. This means that you try to simulate the steps the computer goes through when it runs this routine.

You might draw up stationery like that at the bottom of the code to help you work through, tracing the code. For your first go at this, assume `printToLimit` has been called with `limit` being 2. Then try 7. Then, see if you can generalise to any value of `limit`. once you can do that, you will be ready to write the comment that should be at the beginning of the routine.

```pascal
printToLimit (limit: Integer) is
== well?
==
==
-- no extraneous space is printed at the end of lines
pre
  limit > 1
const
  MAXCOL: Integer := 5
var
  num: Integer := 1
  col: Integer -- what is this?
do
  loop
    exit on num > limit
    print (num)
    col := 1
  loop
    exit on col > MAXCOL - 1
    num := num + 1
    exit on num > limit
    print (" ", num)
    col := col + 1
  end loop
  print ("\n")
  num := num + 1
end loop
post
  true
end printToLimit
```

<table>
<thead>
<tr>
<th>Step #</th>
<th>value of variable named num</th>
<th>col</th>
<th>comments</th>
</tr>
</thead>
</table>

† It taken from our self-assessment web site.
5.5 Problem solving: self-assessment or how do you know when you know and how do you know when it is enough

This is one of the major concerns of students who study in a course based on Problem-based Learning. Obviously, it is easy to get carried away with your particular view of the problem at hand and forget that you have to also pass an examination at the end of the semester. Perhaps you hope to make a fortune selling your beautiful Blue code to the commercial world. That would be nice.

But, you have to keep in mind that the problems you work on in this course are intended as a learning environment. As you work, you need to keep your eye on the overall course goals. To help you do this, we have provided several ways you assess your own progress in the course. Some class time will be devoted to helping you learn more about the very important skill of self-assessment.

The resource we provide to assist you in self-assessment include:

- Course postconditions (in Chapter 2)
- Concept inventories (in the Blue text and back cover of this book)
- On-line self-assessment tasks with example solutions, assessment criteria and an indication of how your judgement of example solutions compared with ours. †
- Exercises in the Blue text.
- Self-assessment tasks in the tutorial classes.
- Tutorial classes where you will practice creating good self-assessment tasks for yourself.
- The Blue environment which, when combined with your own imagination will enable you to create your own self-assessment tasks involving Blue programming.

5.5.1 Concept inventory 1

A concept inventory is simply a list of concepts that you go through, indicating how well you feel on top of these. We will split the concepts into the three main areas of the course. Since all are important, you should make sure you take care with all of them.

Why would that be a good idea?

The concept inventory here includes all the things you should meet in some degree by the end of Problem 1. Even then, we would not expect you to have a deep knowledge of all the concepts. It may look an intimidating list. As the semester wears on, it will transform into an impressive list of the things you have learnt!

† At this point we need to thank the sponsor for these. The creation of this resource was funded by a Science Faculty grant intended to help departments produce support materials for students who want to be able to improve their chances of success in first year courses.
Rate each aspect listed here to show how well you feel you know it:

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

<table>
<thead>
<tr>
<th><strong>Generic aspects</strong></th>
<th>Score</th>
<th>Notes on what to do about this aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>know some of the people in your tute class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>your tutor’s name and login-id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>overall goals of the course</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task for Problem 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assessment for Problem 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>your duties and those of the class tutors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>planning work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>creativity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>listening skills</td>
<td></td>
<td></td>
</tr>
<tr>
<td>awareness of own problem solving style</td>
<td></td>
<td></td>
</tr>
<tr>
<td>when stuck, define simpler problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ability to generalise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>visual thinking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>finding things up in the Blue resources</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Running Blue and using unix</strong></th>
<th>Score</th>
<th>Notes on what to do about this aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>can send an email to class members, yourself, tutor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read email</td>
<td></td>
<td></td>
</tr>
<tr>
<td>change your password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get to CS1001/1901 home page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start the Blue environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>open a project</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copy a project</td>
<td></td>
<td></td>
</tr>
<tr>
<td>create an object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>execute a routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>create a class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>change source code of a class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compile a class, whole project</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove a class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resize a class icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>use red</td>
<td></td>
<td></td>
</tr>
<tr>
<td>change between interface and implementation views</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Running Blue and using unix** (cont)

<table>
<thead>
<tr>
<th>Score</th>
<th>Notes on what to do about this aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- inspect variable values
- breakpoints
- step through code
- example code in `cs1/examples`

**Blue concepts**

<table>
<thead>
<tr>
<th>Score</th>
<th>Notes on what to do about this aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Programming environment - projects, editor, compilation
- Classes and objects - and the difference between them
- creation routine - why needed and what it does
- class interface
- parameters
- creating, invoking, inspecting objects
- declarations for variables
- strings
- integers
- reals
- booleans
- I/O - reading and printing
- Library - the browser, LList
- Control structures
- case
- loop
- if then, if then elseif
- enumeration class
- Correctness: preconditions, postconditions, invariants, assertions
5.6 Scope exercises

class Class1 is

==============================================================================
== Author: Judy Kay
== Version: 1
== Date: March 2000
== Short: Scope - typical, straightforward example
==
==
== The ‘scope’ or ‘visibility’ of an identifier concerns the range in
== which it can be used.
==
== This point of this code is to show a very simple class where the scope
== operates in a way that is very natural - so all else is omitted
==
== Draw labelled scope lines for all the identifiers in the code
== Then answer the questions through the code
==
== Q1: What is the scope of the identifier ‘Class1’. Justify your answer
== in terms of need for ‘Class1’ to be available to a sensible amount
== of the Blue project.
==
==============================================================================
uses
internal
var
  a : Integer
  b : Integer
interface
creation is
  == Do nothing - but declare a single variable

  var
    c : Integer
  do
    -- Q2. Can we use ‘a’ here? Justify your answer.
  end creation

routines
  routine1 is
    == sample routine template
    do
    end routine1
invariant
  true
end class
class Class2 is

== Author: Judy Kay
== Version: 1
== Date: March 2000
== Short: Scope - example 2 - a little nastier
== Slightly nastier example, where some of the scope is NOT intuitive.
== You should NOT plan to write code like the worst of this.
==

uses
internal
var -- ghastly, unhelpful names for instance variable identifier
    a : Integer
    b : Integer
routines
internal_routine is
    ==
    var
        a : Integer
    do
        -- Q1. Can we use the instance variable ‘a’ here? Justify your answer.
        -- Q2. Can we use the instance variable ‘b’ here? Justify your answer.
    end internal_routine

c (a : Integer) is
    == ghastly name for a routine
    var
        b : Integer
    do
        -- Q3. Can we use the instance variable ‘a’ here? Justify your answer.
        -- Q4. Can we use the instance variable ‘b’ here? Justify your answer.
    end c

A (b : Integer) is
    ==
    var
        c : Integer
    do
        -- Q5. Make up some questions about scope at this point. Answer them.
    end A

interface
creation is
    ==
    do
        -- Q6. Make up some questions about scope at this point. Answer them.
        -- Q7. Can I use a ‘Class1’, another class in this project. Justify.
    end creation
routines
B (A: Integer) -> (a: Integer) is
    ==
    do
        a := A
        -- Q7. Make up some questions about scope at this point. Answer them.
    end B
invariant
    true
end class
class Class3 is

Author: Judy Kay
Version: 1
Date: March 2000
Short: More scope - recursion and dynamic scoping
This example is nasty (ie tough to read) on several counts:
- it uses ghastly identifier names - you never would!!
- the documentation is dreadful
- what it does is rather useless anyway

uses internal
var
  a : Integer -- Q1. Critique these names - think of better ones
  b : Integer
  c : Integer

interface creation (
  x : Integer, -- Q2. Write a better comment here
  y : Integer, -- Q3. and here
  z : Integer -- Q4. and here
) is
  == Pretty dull - just initiate an instance variable
pre
  y < z
  a, b, c := x, y, z
  -- Q4. Write a better postcondition
post
  true
end creation

routines count is

==
  -- Q4. what does this do? Write a good comment for the routine
  -- Q5. Trace the execution of this routine, drawing a separate
  -- box for each time ‘count’ is invoked.
const
  A : Integer := c
  a := a - 1
post
  true
end count

invariant
  true
end class
5.7 Example code for problem 2

All the Problem 2 tasks involve building a system that simulates things. If a programmer had to do a task like this, one of the first things they would do is look for examples of similar simulations. After all, with a little thought you could imagine that simulation programs have been written before. You might have even seen some simulation based systems already. Thinking about these might help you think about this problem. (If you look at the highly polished commercial games like Sim* you might also feel intimidated.)

In the `cs1/examples/Problem2` directory, we have one example program for a task that has been set previously. We have also reproduced the code here. The example code was written by a postgraduate student who is an excellent programmer but when he wrote it he tried to tackle the task as if he were a first year student. In fact, it turned out to be rather better than the solutions developed by students. So when you read it, don’t be intimidated but do try to learn from it! At the same time, after gary had thought about the problem, watched his students working on it (no doubt learning heaps from them) it took about 6 hours work to produce the code presented here. We have been able to convince him to leave it as it was at that point rather than doing additional polishing: it leaves room for some tutorial activities involving reviewing the code.

Here is the specification:

Old problem as example - elevator algorithms

Elevators never seem to work according to a sensible algorithm. Here’s the opportunity to see if you can do better.

In this problem you simulate the behaviour of a multi-floor building which uses a number of short-range vertical transport units (well, OK, elevators) to carry people between floors. The Fisher Research Library could be an example.

Individual elevators have a functionality with which it is assumed you are familiar. However, when a number of elevators are used to service the same area it seems that there is a lot of room for improvement in the way these individual units are coordinated. You may want (?) to spend some time in the Fisher Research Library if you are unconvinced of this...

Can you suggest ways in which multiple elevator performance could be improved with respect to user satisfaction? It may help to also think of those scenarios which lead to the greatest degree of dissatisfaction!

By creating an elevator simulator you can easily test and modify your proposals about keeping elevator travellers happy. Of course, you will have already made some modelling assumptions about just what it is that will constitute user happiness in this situation!

Develop the multi-lift simulator and then proceed to use it to explore your intuitions for optimising user satisfaction. Part of your report will be a recommendation (or set of recommendations) based on your experiences.

Starting on the next page are the:

• project comment
• a selected part of the code
• a change log, reporting what happened during the code development
Project Comment

- What is this project?

Simulate elevator(s) in a building, find efficient scheduling algorithms.

What does efficient mean? How do we calculate it? Efficient for what mix of incoming people?

How do we model elevator demand?

We’ll try to minimize average delay and maximum delay, i.e. time between when a person pushes the button and when they get into the elevator.

- What does it do?

Simulate people arriving on different floors, each with a destination. Decide the behaviour of the elevator(s) -- where to move/stop? Record statistics -- average/maximum delay to embark/arrive. Display some neat graphics of elevators and people if we have time.

- How can I run it (what class should I create, what routines should I call)?

Create a Building, call the runSimulation routine.

- Who are the authors?

Gary Capell.

Tricky things:

modelling arrival of new people
how frequently do people appear?
where do they want to go?
We’ll abstract that question away into the Person creation routine. Later on, we can make it more complex.

elevator decision algorithm
It makes no sense to change direction while we’re between floors, but I guess every time we get to a floor we’ll have to choose where to go next--up or down. Also, should we know the exact destination of people waiting on other floors, or just whether they want up or down?

Leave this in a “decision” routine in elevator. Assume we only know for each floor if there are people wanting to go up or down.

We’ll start with the classic "elevator algorithm" -- keep going in your current direction so long as that makes sense, then change direction.

Later on we might worry about express elevators, but they will probably only make sense when we have a non-uniform distribution of people (e.g. lots of people arriving at ground floor in the morning (speaking of which,
the "create Person" routine should therefore know what the
time is.))

We won’t worry about elevators standing still—they
are all always moving.

modelling method -- event queue or global clock
should we make every object "do something” every time
we go through a loop, or have a queue of events, with each
event happening at a particular time?

For now, (actually this would be pretty hard to change)
go with a main queue of events.
Building.blu

Floor.blu

class Floor is

==============================================================================
== Exists to hold lists of people, and to interact with elevators.
==============================================================================

uses LQueue, LPrioQueue, Event, Person, Direction, Building

internal

var
  building: Building
  level : Integer -- ground floor is level 1
  goingUp: LQueue<Person>
  goingDown: LQueue<Person>

--routines

-- <internal routines here>

interface

creation(building:Building, level:Integer) is
  == Creation routine comment here
  do
    this.building:= building
    this.level := level
    goingUp:= create LQueue<Person>
    goingDown := create LQueue<Person>
  end creation

routines

toString -> (s:String) is
  == printout
  do
    s:= building.tally(goingUp.count+goingDown.count)
  end toString

addPerson(p:Person) is
  == add to our list(s)
  do
    if p.goingUp then
      goingUp.enqueue(p)
    else
      goingDown.enqueue(p)
    end if
  end addPerson

getPassenger(direction:Direction) -> (result: Person) is
  == get the next passenger in line who's going in 'direction'
  == Returns 'nil' if there's noone.
  var
    q: LQueue<Person>
  do
    -- my kingdom for a ternary (little C joke)
    if direction = UP then
      q := goingUp
    else
      q := goingDown
    end if
if q.isEmpty then
    result := nil
else
    result := q.dequeue
end if
end getPassenger

peopleWaiting -> (result: Boolean) is
== anybody waiting here?
do
    result := not(goingUp.isEmpty and goingDown.isEmpty)
end peopleWaiting
end class

Direction.blu

class Direction is enumeration

== only two directions, up and down
== maybe later we’ll add stationary

    manifest UP, DOWN

end class
AltaVista search for "arrival time" -> indicates arrival times normally poisson distribution
AltaVista search for +"random number generator" +poisson
    -> function random = poisson(lambda)

%%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
%%% function random = poisson(lamda)
%%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
%%% By Kittiphan Techakittiroj (17 Mar 1996)
%%% %%% %%% Poisson Distributed Random Number
%%% %%% %%% random The random number
%%% %%% lambda The average number of occurred events per time
%%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
random = -1;
a = exp(-lambda);
b = 1;
while b>a,
  b = b*rand;
  random = random+1;
end;

It looks a bit complex for now -- worry about tricky arrival modelling later.

Enhanced project comments -- main decisions were:
  modelling arrival of new people
  elevator decision algorithm
  modelling method -- event queue or global clock

Created main classes - Building, Floor, Elevator, Person

Filled in class comment for Person and Building.
In writing class comment for Building, realized that we'll need a new "Event" class.

Finished rough class comments for all classes, including Event class.
Added "Direction" manifest class for UP and DOWN (duh)

Added a bunch of classes from the library browser to my project:
  LPrioQueue, Comparator - for the priority queue of events
  LQueue - because each floor will maintain a queue of people
  LList - each elevator will have a list of people

Had to create an EventComparator.
Ran into trouble, not sure how to redefine the routines from Comparator.
Wrote Building.creation

ChangeLog

finished project comments.
Thu May 29 12:40:05

altaVista search for "arrival time" -> indicates arrival times normally poisson distribution
altaVista search for +"random number generator" +poisson
    -> function random = poisson(lambda)

%%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
%%% function random = poisson(lamda)
%%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
%%% By Kittiphan Techakittiroj (17 Mar 1996)
%%% %%% %%% Poisson Distributed Random Number
%%% %%% %%% random The random number
%%% %%% lambda The average number of occurred events per time
%%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
random = -1;
a = exp(-lambda);
b = 1;
while b>a,
  b = b*rand;
  random = random+1;
end;

It looks a bit complex for now -- worry about tricky arrival modelling later.
Thu May 29 13:03:16
Wrote Building.runSimulation.
Compiled Building, etc.
Had to add stub routines in Event, Elevator, Person to get Building to compile.
Thu May 29 14:58:40

Tested building creation, removed some infinite loops.
(too used to C for loops)
Thu May 29 14:58:40

Fixed EventComparator to "pass the buck" to event.
Can probably forget about EventComparator from now on.
Added "equal" and "lessThan" as stub routines in Event
Thu May 29 15:10:52

Filling in equal and lessThan for Event, found that access to instance variables in other objects of the same class is not yet implemented.
Thu May 29 15:14:23

Person creation routine
now takes a 'building' parameter,
queries the building for currentTime, number of floors
Added those methods to Building.
STOPPED WORK (for now)
Thu May 29 15:29:30

RESTART
Fri May 30 15:54:33

Made Person and Elevators subclasses of Event
Can generate lots of Person and Elevator events
Fri May 30 16:48:58

people now get added to floors, does very rudimentary display of what’s going on.
Floors, Elevators, Persons can all generate strings of themselves
Fri May 30 16:59:50

Added framework for Elevator event handling
still need elevator to interact with floor, and to choose a direction
Fri May 30 17:09:05

Added a bunch of changes to try to make Elevators interact with floors. Blue just dumped core on me :-(
Fri May 30 17:22:47

And again :-(
Fri May 30 17:27:31

Wierd-- restarting, then recompiling (without making any changes to the source) seems to work, for a little while.
It’s now crashed about a dozen times, and it’s not so fun :-(
Fri May 30 17:35:36

Blue is still crashing all the time, possibly because of mutual 'uses' stuff?
Hopefully elevators should change direction now.
Fri May 30 17:50:20

It seems to run and do stuff, but the display is just
a bunch of print statements, which aren’t very informative.
Lets do some sort of graphical display now.
Fri May 30 17:56:38

Fri May 30 18:16:04

Added a tally to get a visual representation of what’s happening
Mon Jun  2 18:52:02

There looks like abug with elevators going down below the
ground floor. Inspecting the Elevator object lying around,
I can see that there’s a person in the elevator who got in
at 7, with a dest of 2, but was still in at 1. People aren’t
getting out when they should.

The first step is to add an invariant that everyone _in_
the elevator is going in the right direction, i.e. that noone
has missed their stop.
Mon Jun  2 18:53:42

Invariant added, and it does trip: A person with
dest of 4 is still in there at 3 as the lift is going down.

OK, why?
Mon Jun  2 19:04:07

I guess a postcondition of Elevator.service is that everyone
whose destination is this floor is gone.
Adding that now.
Mon Jun  2 19:06:27

Added, postcondition trips too. OK, we’re narrowing it down.
What’s wrong with Elevator.service?
Mon Jun  2 19:10:27

Wierd, tracked it down to Elevator.peopleOut, but
I can’t see what its doing wrong. Thinking...
Mon Jun  2 19:15:16

Found it. I was calling list.removeCurrent, and
then ALSO calling list.next, but list.removeCurrent
also does an implicit cursor advance. Fixed.

Note that I leave the postconditions and invariants in.
Mon Jun  2 19:30:37
Sample examination

THE UNIVERSITY OF SYDNEY

FACULTIES OF ARTS, ECONOMICS, ENGINEERING and SCIENCE

Computer Science 1 (Introduction to Programming, Comp 1001/1901)

June 1999  Time allowed: 3 hours

Instructions

• Answer all six (6) questions.
• The number of marks for each question is shown against its number. These add to a total of 100.
• For some questions there is a suggested length for your answer. This indicates that an excellent answer, earning full marks, can be written in that length.
• Start the answer to each question on a new page of the answer booklet.
• Clearly label each answer.
• The first five questions will be marked primarily for the correctness of the answer.
• The last question will be assessed on clarity, elegance, simplicity and rigor in addition to correctness.
• This examination is open book: you need to use class materials from this course.

This is a sample examination that is intended to help you prepare for the actual one. Comments like this are in italics and explain things about the purpose of the various types of questions and the way they will be marked.

This examination was designed to test the postconditions of this course. See page 8 of the Blue notes.

Note that this sample is longer than the real examination. It includes several examples of some of the types of question that you should be able to answer.

The examination assumes you have all the materials for this course, including all notes and your own code.

Marking: The exam will indicate the relative marks for each part. Do use that to plan your time as there are some parts of questions that are quite difficult and worth the same marks as some easier parts. Make sure you get through all the parts of the paper you can.
**Question Type 1.**

This is your chance to show that you can read and understand small segments of Blue code. You may need to trace the code, working through one line at a time. In other parts of the question, you may need to reason about the code at a slightly higher level as in the examples on this paper.

This style of question may ask you to state what the output of a code segment is, ask that you improve the names of identifiers, give comments that would be helpful, write good preconditions or postconditions.

i. Write a comment that describes what this routine does. Suggest better names for the identifiers `doit` and `y` and briefly explaining your choices.

   ```blue
   doit (y: Integer) is
   ==
   var
   x : Integer
   do
   x := 0
   loop
     exit on x = y
     x := x + 1
     print( x, "\n" )
   end loop
   end doit
   ```

ii. When called with `Classname.domore(30, 70)`, what does this print?

   ```blue
   domore (bound1: Integer, bound2: Integer) is
   ==
   do
   x, y, z := 0, 0, 0
   loop
     exit on x = bound1
     x := x + 1
     loop
       exit on y = bound2
       y := y + 1
       z := z + 1
     end loop
   end loop
   print( z, "\n"
   )
   end domore
   ```

**Marking:** Full marks for part i if the ‘comment’ indicates you really understand what the code does, the variable names you use indicate what the variable does in this code and you also explain the choice well. (Note that in the first segment ‘x’ cannot be given a more helpful name as its real meaning is not clear), Full marks for part ii if you get the right value: miscalculate the loop by one earns 75%.

To study for this question, your best approach is to read examples of Blue code carefully. An excellent source is code in `cs1/examples`, because you can read the code, then experiment with it to check whether you really understood it. Some small examples you could explore are `timer.bp`, `database.bp`, `atm.bp` and `drink_machine.bp`. 


Question type 2.

Here you show you know how to ‘use reference manuals and other printed material to find information about Blue’. Questions may look like this:

A friend has found a program that has the following:

\[
\text{command := readStr.strip.toLower}
\]

The friend asks you how to work out what this does and what it all means.

i. State each step you would take to help answer this question using the resources for this course. Take care to state any document you use, each page number you consult and where you find answers.

ii. According to those resources, what other similar routines are there? List the names of these routines and state what makes them similar.

iii. Explain how you would write a simple class to experiment in using these routines. As part of that explanation, write the routine with code using the line above. Then describe how you use that to experiment with learn about these routines.

Marking: Each step will get full marks if you explain it clearly and it is reasonable. The last part of this question would be well done by writing a routine which includes the statement as well as other code required to make it work. You would also describe the way that you could check the effect of this line, either by writing the relevant strings, or by using the Blue environment's tools to inspect strings.

Question type 3.

The main goal here is to assess your skills in planning and managing a programming project using the various approaches treated in the course, such as programming by contract and planning learning and programming as well as evaluating plans as you did in the progress reports. It also assesses your ability to ‘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’.

Imagine that it is week 5 of this semester. You have recently formed a group to work on Problem 2 and have chosen a problem which requires you to simulate an ATM which can be used for all the usual ATM functions, like transferring money between accounts, get account balances, withdraw and deposit money, get account balances.

i. One member of your group suggests that that each class should be tested by a person other than the author. Do you agree? Justify your answer.

ii. The group decides that one important class is the account which maintains the account owner’s details, all transactions for the last month, current balance. How would you apply programming by contract to the discussions about this class?

iii. Suppose that you are allocated the job of writing account by week 8. Write a plan for developing the code to the point where it can be integrated with the rest of the system by week 8. Assume that at this stage you do not feel that you know enough Blue to write the code yet. Explain the purpose of each step.
This relies on learning from the experiences of the course. The marking for the first part is mainly for good justification. Part ii requires you to show you understand what 'programming by contract' means in terms of the particular actions listed. The last part should build upon your experience of planning your learning of Blue, developing code, testing it, using code reviews and integrating it with the group’s project: marks are for the plan and the explanations with more weight for the latter.

**Question type 4.**

This mainly tests if you can ‘read and evaluate a class implementation in terms of modularity, code independence, class interfaces, class relationships, inheritance, cohesion, coupling, overloading;

This question relates to a given class interface. For example, look at the examples in `~cs1/examples/maze.bp` and look at the class interface for Parser.blu.

i. During a code review, a reader suggests that the `getCommand` routine could be made more general is it returned a `Command` and an `LList` of `Item`. That way, we could extend the game to allow commands to operate with more than just one item. Rewrite the postcondition for this form of `getCommand`.

ii. Write the code that a calling routine could use to check whether `getCommand` had got a single word command (with no items).

iii. Describe the degree of coupling between the calling code and `getCommand` in its original form and the form suggested above.

iv. Is `getCommand` cohesive?

v. Design three sets of test data that each test different aspects of the routine. For each test set, state the input, expected output and explain for the purpose of the test.

vi. When you get past reading the interface and look at the actual code of this routine, a friend suggests that it is silly to have the variables `ch` in `getWord` and `word1` in `getCommand` - they suggest it would be more efficient to simply have an instance variable called `ch` that was used in both places. Explain how you would help your friend to see why this would be a bad idea. At the same time, introduce them to the notion of overloading identifiers and coupling.

*Marking: Full marks go to convincing answers - ones that correctly answer the question without other extraneous statements.*

**Question type 5.**

This tests if you can write small but complete routines

Write **routines** to perform each of the following tasks. Take care to choose suitable identifier names, make appropriate use of parameters and return values, and write the precondition and postcondition.
i. Given a string containing a sentence, the routine should print it with all upper case letters converted to lower case. For example, "Here I am" becomes "here i am.

ii. Given an Integer, the routine should print the numbers 1 to N. (eg. for N 6, it prints 1 2 3 4 5 6)

iii. Given N an Integer, the routine should print the lower triangle of numbers 1 to N. (eg. for N 5, it prints

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

iv. more examples in self-assessment materials eg.....

Marking: We will assess answers on whether they indicate understanding. Most of the marks will be for the major ideas needed for the task. Tiny details will incur little or no penalty.

Question type 6.

This question enables you to show you can write a small class. To make this more manageable, you will be asked to do the job in stages and may not have to write the whole of the class.

This question is about Allocations, a class that maintains the collection of student preferences for university entry.

Each student has a name, identification code, TER (in range 15 to 100) and a list of up to ten institution codes. The first institution code is the student’s first preference, the next is their second preference and so on. For example, one student has just two preferences and has the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Nicholas Negroponte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id code</td>
<td>JH19712345</td>
</tr>
<tr>
<td>TER</td>
<td>93.1231</td>
</tr>
<tr>
<td>Institution Codes</td>
<td>SydneyBCST, SydneyScience</td>
</tr>
</tbody>
</table>

and SydneyBCST is the code for their top preference.

The Allocations class has a routine next_student that returns the details of student who has the highest TER and has not yet been allocated to a course. So, the student with the top TER is the first one returned. On the next call, next_student should return the details for the student with the second highest TER. the next call returns the third and so on.

i. Which of the standard collection classes is best suited to representing the collection of preferences for each student.

ii. Which of the standard collection classes is best suited to representing the collection of students.
iii. Write the interface for Allocations as it has been described above. Write the interface for the class that represents each student’s details.

iv. Write declarations for the instance variables that would be needed for this class.

v. Write code for creation of this class.

vi. Write the uses and var declarations for the For any user-defined types you need, include a comment explaining them.

vii. Suppose that you were writing this Allocations class as part of a larger project.
### 5.9 Concept inventory 2

**Object-oriented programming**

Those concepts marked shown in **Helvetica** have names used in Blue. The same concept applies in other Object-oriented programming languages but under different names.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Rating</th>
<th>Comments on action to take next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projects, editor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compilation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classes and Objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td>classes versus objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scope</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>objects defined v instantiated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mainfest v dynamic classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>references</td>
<td></td>
<td></td>
</tr>
<tr>
<td>undefined, nil references</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic constructs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>declarations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple data types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string, integer, real, boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O on basic data types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reading, displaying</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Library classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>library browser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLList, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control constructs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conditionals, selections</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correctness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pre, post, invariant, assert</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Software engineering concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Rating</th>
<th>Comments on action to take next</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code reuse</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>modularity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>code independence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class interfaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>relationships</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cohesion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coupling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>overloading</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Object-oriented design</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td>classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Code quality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>code reviews</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pre- post conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invariants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assertions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>debug writes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>normal cases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boundaries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalence partitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Style</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>documentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>white space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>self-documenting code</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Group co-ordination</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>programming-by-contract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>planning</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. General resources - and pragmatics
6.1 Getting started with Unix
6.2 Second steps with Unix

page 1
6.3 Do I need my own computer? or How do I get extra computer time?

You do not need your own computer to do this course.

On the other hand, people with access to a computer which runs Blue can work at home at their leisure. You can afford to be a little less organised. You may also prefer the quiet of home for thinking about what you are doing. You may be able to download Blue and install it on your computer. Look at the CS1 web page.

Others will need to be sufficiently organised to come and use the equipment on campus. You can check our opening hours at the web site attached to the class web page (see back page of these notes). Generally, we start the semester with shorter hours and only open on week days. Later, as the pace picks up, the week day hours increase and we open on weekends. Those who do not have a home computer often report a sense of deprivation because they are aware that a lot of the class does have one.

On past experience, many of the students who do best in Computer Science courses do not have a machine at home. These students make sure they plan out work carefully, think about things away from a computer and then come in to uni and work very productively. These people also tend to start work early, not leaving lots till the last minute.

Also based on past experience, many students with easy access to a computer at home spend many hours ‘working’ on computer science, often producing remarkably little. If you do have a computer, try not to fool yourself that all the time you spend playing games, reading news, surfing the internet, sending mail ... is work. Also, you will need to make sure not to leave things to the last minute on the assumption you can always work on your home machine.

Everyone will need some time to work on the computers in the Computer Science department, outside scheduled class times. There are three important things to know about this.

• Start assignments early, as the Computer Science computers always have lots of spare capacity early in the piece.

• Use the booking program (see the Unix information or try reading the on-line manual with \textit{man book}) to book time on a terminal - this is essential for planning your time. Dropping in randomly in the hope of finding a free terminal is chancey and very low odds of success near deadlines.

• Do not leave things till the last minute. (OK, this is repetition but the point is very important - ask higher year students.)

In summary, you do not need a computer of your own. However, it can be very handy, especially if you are unfortunate enough to still have a lot of work to do right near the deadlines and if it happens that other students are in the same boat.
6.4 Dialing in
6.5 Pause gymnastics
The semester as a whole - deadlines and marks breakdown

The following summaries will make more sense if you first read them with the associated materials in the main text. However, you need to refer to them throughout the semester.

This summary is from Page 11.

<table>
<thead>
<tr>
<th>Week</th>
<th>Assessment Task</th>
<th>Percentage</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Group</td>
<td>Individual</td>
</tr>
<tr>
<td>1-4</td>
<td><strong>Problem 1</strong></td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>1</td>
<td>Reflective diary, plan and progress data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Individual certification</td>
<td></td>
<td>10*</td>
</tr>
<tr>
<td>5-12</td>
<td><strong>Problem 2</strong></td>
<td>†††</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Stage 0 - Plan and progress data</td>
<td>20</td>
<td>†††</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Stage 1 - Prototypes, Acceptance tests</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Stage 2 - Certification</td>
<td></td>
<td>10*</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Stage 3 - Individual code</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>Stage 4 - Demonstrations</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Stage 5 - Reflective diaries</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td><strong>Practical work totals</strong></td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td><strong>Written Exam</strong></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td><strong>Grand total marks</strong></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>
7. Summary of assessment

The following copies of the assessment sheets are to be attached to your submissions during the semester.

Problem 1 assessment

This copy is for you to hand in with your folder. Note that no fractional marks are awarded.

Marked by you tutor, in 3-hour workshop, by week 4

On the marksheet you submit, annotate the marksheet to indicate where the marker will find aspects with Location.

Criteria for a pass and certification in Problem 1 (5 or 6 out of 10)

You must achieve a Pass before you can begin Problem 2.

Required criteria for the grade Pass

The submission must have all of the following elements.

- It is in a folder with the code you have written, the test report and statements, all firmly secured and well presented.
- 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 completed self-assessment for the problem must be attached to your folder.
- Your code should be demonstrably able to do the job it is supposed to do: it must work.
- The code to be assessed should be no more than 4 pages long. Location:
- Your submission must include one whole class. Location:
- Your code must make use of at least one loop. Location:
- It must make use of at least one if-statement. Location:
- It must make use of at least one LList. Location:
- 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 1 page.
- 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 criteria for the grade Pass

At least three (3) of the following must hold:

- 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 (e.g., 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, postconditions and class invariants. Location:

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 (7 out of 10)

All criteria for Pass as well as all * items and at least 3 more:

* Must be handed in on time: no later than week 4’s 3-hr workshop.

* Testing is convincing (within 1 page 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.

* All the Blue control structures should be appropriate for the task.

* There should be no 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.

* Your code includes references to some sources you used (for example, if it is based on information from an accounting book, it should give the title, author, publisher, year, pages you used - information from a bank could be the details of the document - if you used some of our Blue resources, state the page/url and how it was used)

* Good attempt at correctness support: preconditions, postconditions and class invariants. Location:

Criteria for a Distinction ++ (8 or more out of 10)

All Pass and Credit criteria plus some of following (for 9+, all starred items, for 10 all items):

* Code should be clear and simple.

* Code does something interesting and challenging.

* More sophisticated Blue aspects used (e.g., nested loops, more than a single LList).

* Each test should test a different aspect of the class, the ‘purpose of the test’ should make this clear and the testing should be elegant and convincing.

* Each routine should do a well defined task (the same one described in its interface comment), have good choice of parameters and good identifiers.

* Each class should be well designed, with good choice of routines and instance variables.

Note that if you are in COMP1001 and would like to be considered for COMP1901, you can demonstrate your ability on Problem 1. If you can achieve a high grade and on work submitted by your 3-hour workshop in week 3 (repeat three), your tutor will discuss transferring you to COMP1901. (Note: even if you have no aspirations for COMP1901, you are most welcome to complete Problem 1 early. Your tutor will be delighted to be able to mark it ahead of the rush in week 4.)
Problem 2 - Stage 1 - Prototypes plus acceptance tests

To be marked by your tutor, in the 3-hour workshop no later than the week 7.

Criteria for a Pass on Stage 1 of Problem 2 (5 or 6 out of 10)

Required criteria for the grade Pass - The submission must have all of the following elements.

☐ A folder containing all required parts of the work firmly secured and well presented.

☐ It must begin with a signed statement that *every member of your group has read the final form of the whole submission* and this must be signed by each member of the group.

☐ The functional prototype part of the report will have the project diagram for the code written and up to 1 page of example input/output.

☐ The structural prototype part of the report will have the project diagram and the class interface view of the major classes.

☐ The Acceptance Tests part of the report will state the tests that you would expect to present at the final demonstration of the semester and it will be no more than 1 page long.

☐ The functional prototype demonstration provides one scenario of the system operation: the marker will observe a running Blue program that shows what the complete system would do in one well-chosen case.

☐ The functional prototype gives a good indication of what the final system will do.

☐ The functional prototype code includes a comment explaining how the final program will solve your groups’ problem;

☐ The structural prototype has at least the main classes needed to make the simulation actually do its job, simulating the activity required for your understanding of your chosen problem.

☐ The structural prototype design includes comments stating which group members are responsible for each class.

Criteria for a Credit (7 out of 10)

All the Pass criteria plus all starred (*) items and at least 3 other items below.

☐ * The complete submission must be handed in on time.

☐ The functional prototype gives a clear idea of what the group intends to do.

☐ The functional prototype makes it quite clear what the group intends the full program will do and it is quite clear that this will address the problem chosen.

☐ The structural prototype design looks sound, clear and simple.

☐ The structural prototype design gives the most important classes in enough detail for the marker to see the instance variables and the interface routines and this is mainly well done.

☐ The acceptance tests are well chosen and convincing.

Criteria for a Distinction ++ (8 or more out of 10)

All criteria for Credit plus 1 of the following for 8, 3 for 9, all for 10.

☐ The structural prototype design is elegant.

☐ The structural prototype design is at a level that the future development is a simply matter of upgrading current classes.

☐ The detailed design of the classes in the structural prototype has good choice of instance variables, the interface and correctness support to match.

☐ The acceptance tests are simple, elegant, clear, minimal.
This side is attached to the folder
Problem 2 - Stage 2 - Code certification

To be marked by your tutor, in the 3-hour workshop no later than the week 9.

Criteria for a pass and certification on Stage 2 of Problem 2 (5 or 6 out of 10)

You must earn a Pass if you are to be entitled to the group marks.

Required criteria for the grade Pass

The submission must have all of the following elements.

☐ It is in a folder with the code you have written, the test report and statements, all firmly secured and well presented.

☐ It must begin with a signed statement that the code you have submitted was entirely written by you (and if you needed a significant amount of help for any other aspect of the code, you should explain that near this signed statement).

☐ It must have a statement that the code is part of the group’s project and this must be signed by you and all members of your group (where this can include, for example, a class that tests out another class that will contribute to the final project - such a testing class may not actually be part of the final project but does contribute to the group’s work; another acceptable example is a class which explores one way to do part of the problem, while someone else writes another class which explores another way - only one will be part of the final project but so long as the group agrees that both approaches should be explored, both are contributions to the group project and this situation should be clear from the group planning sheets; yet another example is where the group decides that one part of the program is critical to the success of the project and therefore asks two group members to work on independent versions - here too, only one will be part of the completed project but both are part of the group’s agreed work on the project).

☐ The code to be assessed must be clearly indicated, perhaps with a cover sheet stating where the marker should look and if this is a particular routine, it should be clearly marked with highlighter along the edge of the page.

☐ The code to be assessed should be no more than 5 pages long.

☐ Your submission must include two whole classes.

☐ Your code must make use of several loops, including at least one nested loop, several case and if-statements: a good testing class, which tests out another class, can easily need these.

☐ It must make use of at least two collection classes.

☐ You must submit a Test report which list the tests you did to convince yourself that it works correctly: maximum length is 1 page.

☐ In your 3-hr lab you must demonstrate that your code works by doing these tests (maximum time for this demo is 5 minutes).

Criteria for a Credit (7 out of 10)

All criteria for a Pass plus * items and 2 more of the ☐ items following:

☐ * The complete submission must be handed in on time.

☐ * There is an attempt to write preconditions, postconditions and class invariants.

☐ Good style and documentation - meaning

• Layout must be consistent and clear, with indentation showing code structure.

• 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.)

• Each class should have a comment stating where you found the ideas and code that helped you write that class (eg the text, reference, on-line examples, other members of your group,.....).
• Each routine should have comments explaining what it does in terms of its parameters and return values.
• Helpful comments explain each of the major instance variables.
• Names for classes, routines, parameters and variables should be helpful.
• Good use of `const` identifiers to avoid magic numbers and to ensure the code can readily be modified to do slightly different tasks.
• Good use of `enumeration` classes as appropriate, to make clearer values for data.
• There should be helpful comments through the code as needed.
• The spelling is good and overall English expression in comments is pleasant to read.
• Presentation of the code means the reader can understand it with minimal effort - it should be pleasant to read, with self-documenting code style and judicious use of comments.

☐ Good design - meaning
  • All the Blue control structures should be appropriate for the task.
  • There should not be tedious code. (If you need to write a swathe of print statements, make sure these are not a significant part of the code you submit for assessment.)
  • There should not be duplicate or near-duplicate code. (So if you could write a single routine which could do the task(s) of several repeated pieces of code, you should write that routine, reducing the overall length of your program).
  • Routines are local as appropriate, none available at interface unless necessary.

☐ Testing is convincing (within 1 page limit), stating: the purpose of the test; the input for the test; the expected output or behaviour; observed behaviour; presentation is clear.

☐ Checks on all user input to ensure that it is ‘reasonable’ and then the code should handle unreasonable data gracefully, perhaps inviting the user to enter the data again.

Criteria for a Distinction ++ (8 or more out of 10)
All Credit criteria plus * items, and at least 1 more for 9, all for 10. of the following:
  ☐ * Code does something interesting and challenging.
  ☐ * Good attempt at correctness support: preconditions, postconditions and class invariants.
  ☐ Code should be clear and simple.
  ☐ Each class should be well designed, with good choice of routines and instance variables.
  ☐ Test page is a delight to read, clear presentation, spelling correct, within space constraints, each tests assess a different aspect.
Problem 2 - Stage 3 - Individual Code

To be marked from the report and a demonstration in the 3-hour workshop no later than the week 12.

Criteria for a pass in on Stage 3 of Problem 2 (10-12 out of 20)

Required criteria for the grade Pass
The submission must have all of the following elements.

☐ It is in a folder with the code you have written, the test report and statements, all firmly secured and well presented.

☐ 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).

☐ It must have a statement that the code is part of the group’s project and this must be signed by you and all members of your group.

☐ The code to be assessed should be no more than 6 pages long, and you must indicate which parts, if any, were submitted for Stage 2.

☐ It must perform both input and output on files.

☐ You must submit a Test report which list the tests you did to convince yourself that it works correctly: maximum length is 1 page.

☐ In your 3-hr lab you must demonstrate that your code works by doing these tests (maximum time for this demo is 5 minutes).

Criteria for a Credit (13-14 out of 20)
All criteria for a Pass plus * items and at least 2 of the following:

☐ * Must be handed in on time.

☐ * Good attempt at correctness support: preconditions, postconditions and class invariants.

☐ Good style and documentation (as described for Stage 2)

☐ Good design (as described for Stage 2)

☐ Testing is convincing and clear (within 1 page limit), stating: the purpose of each test; its input; expected output or behaviour; observed behaviour.

☐ Checks on all user input well done.

Criteria for a Distinction ++ (15 or more out of 20)
All criteria for a Credit plus * items and 1 mark extra for others:

☐ * The amount and overall quality of the code written goes beyond meeting the minimal requirements listed above.

☐ * Code should be clear and simple.

☐ 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.

☐ Excellent correctness support, thorough and well formulated, and no instance variables used for interface (pre/postconditions).

☐ Delightful presentation.
This side is attached to the folder
Problem 2 - Stage 4 - Group demonstration and poster
To be marked by your section leader in the 3-hour workshop in week 13.

Note that the time limit for the demonstration is **10 minutes**. Any group that goes beyond this will be stopped at 12 minutes, regardless of what has or has not been demonstrated.

**Criteria for a pass on Stage 4 of Problem 2 (10-12 out of 20)**

**Required criteria for the grade Pass**
The submission must have **all** of the following elements.

- The group begins the demonstration by stating what the project is intended to do.
- And the demonstration indicates that it does this.
- Each member of the group participates in the demonstration in some way.
- Demonstration is pleasant to attend.
- The group explains how the system developed actually solves the problem and the demonstration indicates that it does achieve a significant part of that role.
- The poster clearly states the goals, how the system addressed the problem, the strengths of the system, shows examples of the system in operation, includes a list of the people in the group.

**Criteria for a Credit (13-15 out of 20)**
All criteria for a Pass plus * item plus 1 for each extra:

- * Demonstration within 10 minute limit.
- The demonstration shows a convincing range of the system operation.
- Presentation is well structured and easy to follow and the important points are clear.

**Criteria for a Distinction ++ (16 or more out of 20)**
All criteria for a Credit plus * item and 1 mark for each extra:

- * The demonstration appears well planned and rehearsed.
- Presentation demonstrates the system convincingly, showing how it is used for to address the central problem.
- System as demonstrated has some interesting aspects that go well beyond the basic task.
- Presentation is entertaining.
- Poster is clear, elegant and visually appealing.
This side is attached to the folder
Stage 5 - Reflective diary

Criteria for a pass on the reflective diaries (5-6 out for 10)

Required criteria for the grade Pass
For each week of Problem 2 (weeks 5-12):

- a summary of the work done in that week;
- a reflective assessment of that week’s work, how well, in hindsight, you consider that you worked that week: perhaps you wasted time at the terminal because you had not read enough materials; perhaps you wasted time because you tried to do too much at once, rather than breaking the task down and solving smaller parts; perhaps you were hampered by problems interacting with other group members; perhaps you did not work hard enough...

Criteria for a Credit (7 out of 10)
All criteria for a Pass plus * item and one other:

- * Reflective diary submitted on time
- Attachment of supporting documentation for claims about work done (eg attach personal plans for the semester, copies of exploratory code written)
- Diary shows insight into successes achieved as well as reasons for problems and how to address them.

Criteria for a Distinction ++ (8+ out of 10)
All criteria for a Credit plus one for each of the following:

- Diary shows how you assessed your progress on the project and on learning the course goals;
- Diary shows how testing strategies did or could have ensured smooth progress of the project
- Diary shows insight into how demonstrable achievements for the project were or could have been planned for each week.
This side is attached to the folder
## 8. Summary of useful Unix facilities

<table>
<thead>
<tr>
<th>Command</th>
<th>Notes for your use - especially examples of syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
<td></td>
</tr>
<tr>
<td>passwd</td>
<td></td>
</tr>
<tr>
<td>quota</td>
<td></td>
</tr>
<tr>
<td>search-and-destroy</td>
<td></td>
</tr>
<tr>
<td>mv</td>
<td></td>
</tr>
<tr>
<td>cp</td>
<td></td>
</tr>
<tr>
<td>rm</td>
<td></td>
</tr>
<tr>
<td>ls</td>
<td></td>
</tr>
<tr>
<td>ls -l</td>
<td></td>
</tr>
<tr>
<td>cd</td>
<td></td>
</tr>
<tr>
<td>pwd</td>
<td></td>
</tr>
<tr>
<td>mkdir</td>
<td></td>
</tr>
<tr>
<td>rmdir</td>
<td></td>
</tr>
<tr>
<td>chmod</td>
<td></td>
</tr>
<tr>
<td>more</td>
<td></td>
</tr>
<tr>
<td>less</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td></td>
</tr>
<tr>
<td>sdiff</td>
<td></td>
</tr>
<tr>
<td>ghostview</td>
<td></td>
</tr>
<tr>
<td>lpr</td>
<td></td>
</tr>
<tr>
<td>lpq</td>
<td></td>
</tr>
<tr>
<td>red</td>
<td></td>
</tr>
<tr>
<td>ispell</td>
<td></td>
</tr>
<tr>
<td>netfile</td>
<td></td>
</tr>
<tr>
<td>netget</td>
<td></td>
</tr>
<tr>
<td>tkman</td>
<td></td>
</tr>
<tr>
<td>apropos</td>
<td></td>
</tr>
<tr>
<td>man</td>
<td></td>
</tr>
</tbody>
</table>
9. Help and resource summary

General help:

Help desk Madsen lg44, 12:2 and 5-5.45 each working day of semester.

People at the Help Desk can help with many things, administrative and technical. Even if they cannot help, they can start the process of finding people who can and use electronic communication to pass on information once they have it.

Blue resources:

*An Introduction to Computer Programming with Blue*,

*Introductory Programming and Introductory Computer Science Reference Manual*,
Michael Kolling, 1998 or 1999 printing.

Blue resources http://www.cs.su.oz.au/~blue

Example code in `cs1/examples`


Self assessment


Other resources:

The University of Sydney home page
http://www.usyd.edu.au/

Department of Computer Science home page

The University Code of Conduct for ISPs

The university has a number of sources of help for many problems.
Check out the union information desk about such services.
Or look around on the web by starting at the Sydney University home page.
4.2.2 Problem 2 - Stage 1 - Prototypes plus acceptance tests  27

4.2.3 Problem 2 - Stage 2 - Code certification  28

4.2.4 Problem 2 - Stage 3 - Individual Code  30

4.2.5 Problem 2 - Stage 4 - Group demonstration and poster  31

4.2.6 Stage 5 - Reflective diary  32

5. Eclectic set of resources to support tutorial work++ ....................................................................... 33

5.1 Problem solving skills : individual planning ................................................................. 33

5.2 Group problem solving : creativity ......................................................................................... 37

5.3 Problem solving skills : problem solving style, resources, listening ...................................... 40

5.4 Problem solving skills : tracing and generalisation ............................................................. 43

5.5 Problem solving: self-assessment or how do you know when you know and how do you know when it is enough ................................................................. 44

5.5.1 Concept inventory 1  44

5.6 Scope exercises ....................................................................................................................... 47

5.7 Example code for problem 2 ................................................................................................... 50

5.8 Sample examination ................................................................................................................ 58

5.9 Concept inventory 2 ................................................................................................................ 64

6. General resources - and pragmatics ................................................................................................. 66

6.1 Getting started with Unix ........................................................................................................ 66

6.2 Second steps with Unix .......................................................................................................... 69

6.3 Do I need my own computer? or How do I get extra computer time? ..................................... 75

6.4 Dialing in ............................................................................................................................... 76

6.5 Pause gymnastics ..................................................................................................................... 80
7. Summary of assessment ................................................................................................................... 85

8. Summary of useful Unix facilities ................................................................................................... 97

9. Help and resource summary
Figure 1. Organisational Structure from the COMP-1001/1901 Perspective ................. 15
LIST OF TABLES

TABLE I. Study Commitment ........................................................................................................ 14
Acknowledgements

This resource book builds on the contributions of several of the members of the Comp1001/1901 team and their predecessors. Chapter 1 was largely written by Tony Greening and Chapter 2 is an adaption of his notes. Many suggestions and improvements were provided by kind reviewers of the manuscript. Particular thanks for contributions of content from:

Matt Barr
Alan Fekete
Tony Greening
Jeff Kingston
Nicole Lesley
Juhyun Pak
Nathan Sinclair
Risk Management Office of the University of Sydney

My thanks to Gary Capell for agreeing to allow the use of his code. Thanks also to the students who made helpful suggestions in the previous instantiations of the course.

Preface

This resource book will be used through the semester. However, you should make it a priority to skim the first two chapters early in the semester. They give the overall picture of what you must do for the course.

The third chapter has Problem 1 resources. This will be the main focus for your first month of work. This problem is a warm up for the main problem of the semester, which is described in Chapter 4.

Both Chapters 3 and 4 have assessment criteria. The importance of reading these should be obvious. But every year, students submit beautiful work that deserved to do well but which misses elements of the published assessment criteria! Make sure this does not happen to you or your group.

Chapter 5 has the resources that will be used in tutorials and lab classes through the semester: you will need to have these available for all such classes. The back cover of this book has room for you to keep track of useful Unix commands and resources for working on Problem 2.

This book should help you learn about programming and creating beautiful solutions to Problems 1 and 2. Please mail comments and suggestions.

Judy Kay (login: judy)
PBL Co-ordinator