A Personal Portfolio covering an ITP Module

David J. Barnes
The University of Kent
d.j.barnes@kent.ac.uk

A Personal Portfolio covering an ITP Module

David J. Barnes
The University of Kent
d.j.barnes@kent.ac.uk

1 Executive Summary

Surely having spent over twenty years teaching introductory programming, there is little left for me to learn either about what to teach or how to teach it? In fact, experience suggests that neither of these is the case. The ongoing debate about objects first, procedures first, functional first makes it clear that there is no general concensus about what to teach in an ITP course; while widening participation at University level and changing learning and study skills of incoming undergraduate students challenge whether we can still teach an ITP course in the same way as we did twenty or thirty years ago.

My personal motivations for undertaking this project arose primarily from a desire to teach well and effectively, and recognition that teaching techniques need to be appropriate to the 'client' population of learners. Unless I reflected on my current practices, I could not be sure that the existing fit was a good one.

This portfolio is a personal exploration of one particular instantiation of an ITP course, taught by me within the Computer Science Department at the University of Kent, UK.

2 Context

2.1 Departmental Context

Founded in 1965, The University of Kent was known for many years as, "The University of Kent at Canterbury" – or UKC for short. Since 2000, however, a campus based in Medway (UKM) has been added and the University prefers to be known now by the shorter form. Nevertheless, most courses are based at a single site, and the course that is the focus of this study is taught entirely at the Canterbury campus. The Department of Computer Science has its origins in the formation of a Computing Laboratory in 1966, out of the Mathematics Institute. Originally providing a dual computing service and teaching role, the Computing Service was eventually made independent of the Computer Science department in the 1990s.

The department expanded steadily in the early 1980s, and again as a result of global increased student interest following the .com revolution in the 1990s. We were early adopters of the Java programming language for teaching, which we integrated throughout our computing-based degree programmes. The main undergraduate degree programmes we offer are Computer Science, Business Computing, Computer Science and Management Science, and Web Computing. Computer Science with Artificial Intelligence begins in September 2006. We also have several taught MSc programmes, such as Distributed Systems and Networks, Pervasive Computing and a Generalist Computer Science MSc.

The department is one that has always encouraged and supported good teaching and teaching innovation. From time to time it has offered teaching prizes, voted for by the students, and latterly it has encouraged staff to enter for the Faculty's learning and teaching award, and the National Teaching Fellowship.

2.2 The Course Context

The course that is the focus of my reflection is *CO520 Further Object-Oriented Programming*. Some context might be necessary to explain why a 'further' programming course is properly an 'initial' course.

CO520 is a 15-credit module delivered in Stage 1 of the Computer Science (and CS variants) programme. It is taken by around 170 full-time students each year. A 15-credit module corresponds to 150 hours of study. This time allocation is intended to cover all elements of the course: teaching, personal study time, coursework, revision and examination. Students take 120 credits each year, mostly in the form of 15-credit modules. A degree at Kent lasts three years (or four years if a student spends their third year on a work placement in industry).

Until 2004, the material taught in CO520 was the second half of a 30-credit module called *Introduction to Object-Oriented Programming*, the name now used solely for the module CO320 which covers the first half of the original material. Most of our CS teaching has recently been revamped so that it is conducted in single term, 15 credit 'short fat' modules like CO320 and CO520.

So I still tend to think of CO320 and CO520 as a continuum of introductory material. CO320 covers the basics of object-based programming (classes, objects, encapsulation, libraries, etc.) and CO520 continues with inheritance structures and larger applications, including those involving GUIs. I provide class support in CO320 and I am the sole lecturer on CO520.

CO520 is delivered at the rate of two lectures per week, with a single terminal-based class in between. Classes have between 12 and 18 students in them, with a class supervisor who is either a lecturer or a postgraduate. Assessment is via three programming assignments, worth 30% of the module mark, and a two-hour written examination paper worth 70%. This ratio is largely historical, and an artefact of the regulation that coursework worth more than 30% must be second marked.

All CS modules are supported by a module-specific web page based around a departmental template [6]. Staff teaching the module have complete control over what is made available via the page in addition to standard information such as module description and assessment details. On CO520 I use a 'message of the day' area to deliver important messages via static text, and this is also mirrored as an RSS feed. Course material consists of lecture slides published in advance of the lectures. I also publish MP3 recordings of lectures immediately after delivery. Both the RSS feed and the recordings are individual decisions on my part, rather than departmental policy.

I also provide a help-seeking web page that enables students to ask questions anonymously and receive answers that everyone can read. This reduces the number of questions I am asked via direct email, and reduces the number of duplicate questions I get asked because students find answers via questions others have asked. I have published a couple of papers in this area [1, 2]. Similar support is offered by my colleagues on their modules.

I make it clear that I am happy to answer questions about any aspect of the course – including assignment questions or 'syntax error' problems. I consider it to be important to respond promptly to questions. While I do accept questions via email, most students choose to use the anonymous page.

3 Aims and Philosophy

I aim to teach with enthusiasm and make my subject interesting. I endeavour to make what I teach accessible to everyone and provide challenges for those who want to take the subject further than the limits of the curriculum. I seek to engage students with the course and to support them in their learning. I am prepared to devote significant effort to help students who are struggling to learn yet keen to do so. My hope is to create independent, self-supporting learners who would ultimately be able to teach me.

I am not an educational psychologist, but I suspect that my teaching style is firmly rooted in the constructivist camp. The following, paraphrasing a couple of sentences in [7], captures something of this:

> I consider the knowledge and experiences students bring with them to the learning task then teach in such a way that students can expand and develop this knowledge and experience by connecting them to new learning.

I believe that teaching programming well is difficult. Until relatively recently, I believed that it should be possible to teach anyone how to program. I am no longer sure that this is true, at least within the limited time and resource constraints of a University course. Nevertheless, I would never give up on anyone who genuinely seeks to learn to do so.

See appendix A for an artefact that represents how some of my philosophy and learning support is expressed in practise.

4 Content

4.1 The Course Content

Appendix B contains details of the formal learning outcomes and syllabus for the course. In contrast, this section contains what might be described as the content at the coal face.

Within the current implementation of CO520, Java is the programming language that is used in practice. CO520 builds on CO320, which delivers an objects-first approach to object orientation, and CO520 assumes that students are already familiar with fundamentals of object-based programming: classes

and objects, fields, methods, constructors, selection and iteration, collections and encapsulation, along with related programming skills such as testing, debugging, documentation and use of code libraries.

The contents of these two modules are tied closely together, mainly because they originally comprised a single thirty-credit module. The content and philosophy of both modules is captured in a textbook authored by Michael Kölling and me [4].

The OO and Java content are covered in the following order:

- Class inheritance via `extends`. Recognising that classes are sometimes similar to one another, in content or function or both. This similarity can often be captured in the form of an inheritance relationship, where subclasses inherit data and functionality from a common superclass. We cover basic pure extension inheritance first of all, and then move on to cases where subclasses wish to alter inherited behaviour using method overriding.

- Further abstraction techniques using abstract classes are covered next. This is presented by recognising that some superclasses have no useful default implementation for some behaviour, or have no independent existence outwith their subclasses.

- The concept of abstract classes is then further explored in terms of interface inheritance, and the idea of multiple inheritance.

- Building graphical user interfaces via the Swing library is now covered. This material builds heavily on the use of class and interface inheritance. Nested classes are also introduced as being the standard way to write event listeners in Java.

- Defensive programming through the use of exceptions and assertions. The importance of checking parameter values is stressed, but also the balance between trusting clients and regarding them as potentially hostile. Input-output is included here as the need to handle exceptions occurs naturally with I-O.

- The course finishes off by looking at the design and implementation of larger applications from an informal specification, using the noun-verb method and CRC cards.

4.2 Why the content is as it is

- Because CO320 covers all the necessary object-basics material, we can kick straight off with inheritance.

- GUIs are covered relatively late because of their dependence upon relatively advanced concepts (inheritance, interfaces, etc.). We believe it is a mistake to try to cover raw details of GUIs early. Using libraries to hide the detail may be a different matter.

- Many courses use I-O early. In fact, I-O is not the easiest topic to cover, particularly at an early stage. One advantage we have in using BlueJ [3] is that input-output is needed until quite late because of the interactivity of BlueJ. Input to objects is possible via direct interaction with parameter lists of methods and constructors.

- Historically, first-year programming courses typically included a large amount of material on algorithmic programming, such as sorting and searching, and implementation of data structures, such as linked lists and trees. We don't attempt to cover most of this in our first year courses.

  One reason is that we believe you don't need to know how to build your own data structures or sort algorithms at this level. More important is that you know how to use libraries of existing components that do these things for you – and probably do them better than you could.

  A second reason is that we don't believe it makes sense to try to cover everything procedural and everything object-oriented in one year. Most students have enough trouble reaching sufficient competence in one paradigm let alone two.

4.3 Influences on the content

One influence was my own experiences of teaching introductory programming in several different procedural-programming languages over about fifteen years prior to the development of the direct ancestor of CO320 and CO520. I was also influenced by Ian Utting's [9] championing of object-orientation for introductory programming at Kent in the mid 1990s. I came to feel quite strongly that 'objects early' was the most appropriate way to teach object-orientation, and my first textbook [3] reflected this. I then discovered the BlueJ environment and the influence of Michael Kölling's ideas served to go even further and realise an objects first approach.

4.4 Delivery content

Students taking CO520 are typically following four fifteen-credit modules simultaneously. There are two one-hour lectures per week on the CO520, and a one-hour class in between the lectures. Each lecture is a mix of presentation using PowerPoint slides, demonstration and programming using BlueJ, and Q&A with the students – interactivity is encouraged. Audio of the lectures is recorded and made available immediately afterwards via the module web page.Terminal-based classes are intended to help students develop practical facility with the associated material covered in the lectures.

5 Instructional Design

5.1 Introduction

In this area, I experienced the value of participating in this commons simply through the process of reflecting on my practice. What exactly do I do when I deliver this course; and why do I do it that way?

Sadly, my conclusion is for this topic as it has been for others: a lot of what I do is simply because that is the way I (and my colleagues) have always done it. However, that is not being completely fair on myself, because from time to time I do adapt the way I do things, or adjust what I do, in response either to a request from my students, or to a need that I perceive. I have tried to reflect that in the 'Adaptations' sections below.

5.2 Lectures

I teach using two one-hour lectures per week – primarily because that is what the system expects me to use and provides a well-oiled infrastructure to support. In lectures I can reach the whole cohort as a single body and convey key issues in a reasonably efficient way. The course is closely based around a textbook [4], which I expect students to read as follow up to lectures.

Lecture content is usually choreographed via a PowerPoint presentation and often also includes demonstration of practical programming using an IDE [3]. Additionally, if there is an assignment in progress, the beginning of the lecture will usually be used to invite and answer questions about it.

The PowerPoint slides for each lecture are made available to students before the lecture to support note taking and reviewing. The slides do not contain much text because that can be found in the course text book.

I do actually believe that the lecture format provides a good way to introduce topics to a large audience, and allow a manageable path to be plotted through a potentially large body of material. I don't believe that much long-term learning takes place within such sessions, but I believe they are a good starting point.

5.3 Classes

I use a single one-hour class per week that is held in a room with PCs. Each class has between 12 and 18 students in it. Attendance is compulsory, recorded and monitored. I provide set exercises for students to work through in order to reinforce the key issues I have been talking about in lectures. The class material is taken directly from this book. If an assessment deadline for this course is near, the students may work on that.

In contrast to a large lecture, this is the level of the course at which it should be possible to observe individuals' understanding of the course material and provide feedback that is customised to individuals. In practice, however, a 50 minute class leaves little time for significant individual input.

5.4 Lecture Adaptations

Over the years I have developed a number of adaptations to my instructional style, in an effort to make the material more accessible to an increasingly diverse body of students.

One major adaptation was the introduction of an anonymous help seeking web page, around 1996. I had often observed that students got stuck with assessments and were either too timid to ask for help or thought asking would

jeopadise their marks. So I created a web page through which any question about an assessment or the course material in general could be asked, and answers provided for all to see. This has been very successful, and adopted by many of my colleagues for their courses. I receive huge numbers of questions each year through it (almost 500 in 2005-06). One potential concern is that providing this sort of help discourages development of students' own problem solving skills. I prefer to think of it as a lifebelt to save someone from drowning, so that you can then send them off to take swimming lessons.

Since 2004 I have been recording each lecture and making the recording available immediately afterwards. Many of my colleagues are sceptical of the value of this, but analysis of web logs and end-of-module feedback suggests that many students use them – both following lectures and during the revision period.

Concerned that some students were struggling with the practicalities of even the most basic programming tasks, I experimented by recording several audio-video presentations of me writing programs (essentially active screen captures with commentary) along the lines of the 'process recordings' described in [5]. These have been well received and more have been requested.

5.5 Class adaptations

In previous years the format of terminal classes has largely been heads-down programming with a supervisor on hand if help is needed. However, largely as a result of the reflection I have undertaken as part of the Disciplinary Commons exercise, I have been changing my views on the way these classes should be run. I was able to try out an alternative idea in this year's CO320 classes, for which I was a supervisor, and then implement this in all the classes for CO520.

An important principal for me is that a student who is stuck should have no need to have reservations about asking for help when they get stuck on a problem. Oftentimes, I would observe students in class struggling for long periods with relatively trivial problems that could easily be solved by a little 'expert' help, and when that expert help was on hand. An extension of this Robinson Crusoe approach is that I would rarely be asked for assistance between classes by my students, and most of my class supervisors observed the same thing.

My feeling was that students were not comfortable speaking out in class to ask for help, and were not comfortable enough with me as a class supervisor to feel able to ask for help between classes. So in my CO320 class I decided to 'sacrifice' some of the limited practical time with in a fifty-minute class to a plenary session at the start. The purposes of this were:

- to give the students a chance to use their voices in the class, in the hope that they would be less inhibited from doing so when they had a question;

- to make it clear that I expected coming to class to be an active experience for them;

- to create a relationship between students and teacher where it was clear that their questions would be taken seriously and they were not going to be made to look silly.

Typically the plenary would review what had been taught in the previous two lectures – asking them to tell me as I was not their lecturer – reinforce the material a bit, and answer any questions they might have about that material, or current assessments.

I felt this worked well enough in CO320, with classes becoming much more active than in previous years. So I decided to use it again in CO520, and to ask all of my supervisors to run their classes in similar ways.

It's not rocket science, but it sent out some important messages to the students, and made classes more effective.

5.6 Concerns

One of my biggest concerns about the current state of my instructional design is that it assumes too much that students are capable of and willing to engage in self-propelled learning. I.e., that following a lecture they will go away and read the associated pages of the textbook ready for the next class; that they will finish off the class exercises before next week in their own time; that they will start assessments in good time; that they will ask for help when they get stuck; that they will learn generic lessons about how to solve problems from the way that they solve particular problems.

Practical experience tells me that that most of my students are not like that.

I don't have any stunning answers for how to solve these problems, but I think my awareness of them makes it more likely that I will find ways to adapt my practice to mitigate some of them, and perhaps do a better job of teaching. The suggested learning plan artefact in Appendix A is one attempt to address the deficiency in self-propelled learning.

5.7 Logbooks

As an inverate logbook keeper, I have long felt that it is useful for students to maintain a logbook as a learning tool for their modules. For a couple of years in the late 1990s, I included the keeping of a logbook as an item of assessment within a first-year software engineering module, but had never included them formally in an ITP course.

Feeling particularly strongly this year about the weakness of basic study skills in many students, I decided to reintroduce the use of a learning logbook in my second-term ITP module. I issued the same guidelines I had used eight years ago [8], but made it clear that the logbook would not be assessed. Nevertheless, they were expected to bring it to every lab session and have their lab supervisor initial and date the point they had reached within it.

I believe that logbooks provide a means to capture things that you learn within a course, but also a means to express and explore the learning and discovery process. The students were advised to note objectives for each lab session,

and then to make reflective notes on the actual outcomes of that session. Class material typically made suggestions for things they might like to record in it. They were also encouraged between labs to articulate within the logbook things that they did not understand and bring them up with the lab supervisor at the next session. I believe that the process of articulation can also be a way of working out the answer for yourself.

Reactions from the students were mixed, but generally positive. Some made too close an association between the logbooks and assessment and could not see how a permanent record might be of use once an assessment has passed. Others enjoyed the freedom of being able to record what *they* felt to be the important topics covered in the course. In a whole-course feedback session towards the end of the year the general feeling was that logbooks should be introduced earlier in the course but there were mixed feelings over whether keeping them should be compulsory – slightly more in favour than against.

In hindsight, it was a mistake to issue the same guidelines as had been used on the software engineering course. I will rewrite the guidelines to be more detailed and more easily applicable to an ITP course. Within the department I will seek to encourage their use in the first term of the first year. My own feeling is that keeping a logbook should not be compulsory.

As an additional use, as the logbooks are checkpointed, we could invite students to submit a logbook as supportive evidence in defending a plagiarism allegation, for instance.

6 Assessment

A few of my beliefs about assessment are that:

- It should be developmental.
- It should be practical.
- It should be achievable.
- It should be easy to assess.
- Students should be able to ask questions about it.

The course is assessed 30% by coursework and 70% by examination. Three programming assignments are each worth 10% of the final mark. The examination is a two-hour written paper which largely also assesses programming skills.The 30:70 bias in favour of examinations means that coursework marking can afford to be fairly light-touch, and relatively generous as an encouragement.

Each assessment [Appendix C] is available for a period of about three weeks. In the weekly classes, developmental exercises are provided but these are not assessed. More material than can be completed within class is deliberately provided in order to reinforce the course message that

programming skills are only developed by ‘doing’ programming, and that this should occupy a sigificant amount of time between lectures and classes.

Students submit programs for assessment and markers are expected to annotate printed scripts with comments and marks, in particular pointing out reasons why marks might have been lost in order to help students improve next time.

‘Sample solution’ style feedback is given within one week of an assignment deadline, and all work is returned within three weeks.

Rather anomolously, and unhelpfully, in my view, students receive no feedback from the examination other than their mark. Under The Data Protection Act they could ask for a copy of any markers comments. In practice, this means that markers don’t actually write much of use on scripts.

I have long been dissatisfied with the fact that most of the assessment of this practical and highly technical course is via pencil-and-paper exercises under closed conditions. I would much rather assess the course wholly via open-book programming tasks. It is my intention to allow the use of any single Java textbook of a student’s choice in next year’s examination.

7 Evaluation

7.1 Feedback mechanisms

CO520 is delivered at Stage 1 (first year) to most of its students. An informal feedback session is run mid-term to gather input from students about how the current set of four modules are going. In a plenary session, students are invited to make assertions about each module, e.g. “There is too much coursework in CO520”. At the end of the session each then votes secretely on each assertion on a five-point scale (agree strongly, agree, etc.). Free-form comments also encouraged. Results of the feedback are published to the students shortly after the meeting and any major issues are discussed by a departmental executive group covering Stage 1.

All modules within the University are required to conduct anonymous questionnaires at the end of the module. A number of standard questions are asked, such as adequacy of library provision and computing facilities, but otherwise staff on the module have control over the questions they ask (Figure 1). Free-form comments are also encouraged (Figure 2). Questionnaire results are included on each module’s web page and in an annual module monitoring report (AMMR).

Valid answers were - 1: Agree strongly; 2: Agree somewhat; 3: Undecided; 4: Disagree somewhat; 5: Disagree strongly.

- Before the course began I could already program confidently in Java.
  range 1..5, mean 3.08, S.D. 1.51

- I think the course has been interesting.
  range 1..5, mean 1.90, S.D. 0.85

- The course resulted in a worthwhile improvement in my knowledge and grasp of the subject.
  range 1..4, mean 1.88, S.D. 0.95

- I think the overall workload has been reasonable.
  range 1..5, mean 1.92, S.D. 1.02

- The assessments helped me to understand the course.
  range 1..4, mean 1.81, S.D. 0.75

- The web page for the course was adequate.
  range 1..4, mean 1.54, S.D. 0.68

Figure 1: Sample of end-of-module questionnaire results

- The GUI stuff was very good, the stuff covered in the lectures was interesting. It was also good to have a choice of what we could do on the GUI assessment, as all the other assessments followed a specification instead.

- The first assignment took longer than expected to be marked and returned by the class supervisor.

Figure 2: Sample free-form feedback from end-of-module questionnaire

The AMMR also contains statistical data on coursework and examination performance including, where appropriate, comparisons with previous years results. Module conveners are asked to comment on any significant differences in performance between this year and previous years, and to respond to significant issues raised in student feedback.

Anonymous question-asking pages are provided throughout the module. These support students in asking any question they wish about course material (Figure 3). Students submit questions via an anonymous web interface, and a form interface for lecturers allows them to provide answers as well as keyword and cross-reference indexing. As assessments in CO520 are often developmental in nature, and important use of these pages is to answer assignment-related questions.

Question 327:

Is there a way of making the program wait a few seconds until executing the next line of code? ie if i have the end of a game, is there a way of making the application wait a few seconds before asking if the user wants a new game?

Answer 327:

Yes, you can use the `sleep` method of the Thread class. It involves knowing a bit about *exceptions* which we will be covering in the next few lectures, but here is a canned solution for you:

```
    try {
```

```
        Thread.sleep(length-of-time);

    }

    catch(InterruptedException e) {

        // Not much we can do about this.

    }
```

The *length-of-time* parameter is a number of milliseconds, so 1000 for one second, for instance. The parameter is of type long so you can use very big numbers for very long pauses.

Keywords: Thread.sleep

Figure 3: Sample anonymous question and answer

7.2 QA requirements

As indicated above, each module is expected to complete an AMMR. Where a module's pass rate is above 15% a commentary suggesting reasons for this must be supplied. All AMMRs are considered by the Board of Studies and a programme-level monitoring report summarises issues arising from the AMMRs and programme as a whole.

Our programmes are subject to a periodic programme review (PPR) which involves both internal assessors from other University departments and external assessors from other CS departments.

8 Conclusions

Participation in the Disciplinary Commons has been a valuable experience. The process of reflection it has caused me to engage in has meant that I have changed my practice on-the-fly. Discussing issues with colleagues from a broad range of different environments has been especially helpful in giving me a bigger context against which to consider the learning and teaching issues I face.

9 References

[1] David Barnes, *Students Asking Questions: Facilitating Questioning Aids Understanding and Enhances Software Engineering Skills*. *ACM SIGCSE Bulletin*, 29(4):38-41, December 1997.

[2] David J. Barnes, *Public Forum Help Seeking: the impact of providing anonymity on student help seeking behavior*. In Graham M. Chapman, editor, *Computer Based Learning in Science (CBLIS '99)*. Pedagogical Faculty of University of Ostrava, Czech Republic, July 1999.

[3] David J. Barnes *Object-Oriented Programming with Java: An Introduction*, Prentice-Hall, January 2000, ISBN 0-13-086900-7.[3] BlueJ, Interactive Java Environment, http://www/bluej.org/

[4] David J. Barnes and Michael Kölling, *Objects First with Java - A Practical Introduction using BlueJ, 2nd ed,* Pearson Education, 2005, ISBN 0-131-24933-9

[5] Jens Bennedsen and Michael E. Caspersen, *Revealing the programming process*, p186-190, Proceedings of the 36th SIGCSE technical symposium on Computer science education, St. Louis, Missouri, 2005.

[6] CO520 Module specification. http://www.cs.kent.ac.uk/teaching/05/modules/CO/5/20/

[7] W. Huitt, *Constructivism*. *Educational Psychology Interactive*. Valdosta, GA: Valdosta State University, 2003. Retrieved 2005.11.08, from http://chiron.valdosta.edu/whuitt/col/cogsys/construct.html

[8] Les Johnson (amended by David J. Barnes), *Keeping a Learning Logbook*, http://www.cs.kent.ac.uk/~djb/LOCAL-ONLY/logbook.html [Only accessible within UoK].

[9] Ian Utting, Personal web page. http://www.cs.kent.ac.uk/~iau/

Appendix A: Teaching Philosophy Artefact

Below is part of the weekly suggested learning plan that I created for students as a way to help them see how to use the notional alloted time available to them on CO520.

Suggested Learning Plan

In this section I have outlined a possible learning plan for each week's activities, based around the idea of spending on average about 9-10 hours per week studying the course material. This is only a suggestion. You should feel free to adapt it to your own particular needs and learning style. The key issue is to try to ensure that you spend an appropriate amount of time on this module each week, and that you keep practising the programming skills that are necessary to pass the module.

- Week 19: In Monday's lecture we looked at the different characteristics displayed by layout managers (pp316-322). Be sure that you understand the different ways in which they share out available space, affect the size and juxtaposition of the components that they manage. Use the *examples of different layout managers* to help in your understanding of these. This week's class material will give you the opportunity to build an application that uses layout managers.

  We also looked at different styles of dialog windows (pp325-327). Use the *JOptionPane examples* to explore these. You should read the source code in conjunction with the API documentation for the JOptionPane class.

  In Thursday's lecture we looked at the way in which the actions of the various filters could be captured in a Filter superclass (pp327-332).

  Assignment 2 is due this week, so try to work on that within the first few days of this week rather than leaving it until Thursday or Friday.

- Week 18: This week we will explore how to build GUIs using Java's AWT and Swing APIs. This is covered in chapter 11 of the course text. GUI programming builds on the abstract class and interface material we covered recently, so be sure that you have done all the followup work on that. This week's class material gives you the opportunity to explore various versions of the imageviewer project as we build up its functionality. Study the new features of each version carefully as they illustrate the components, layout managers and event-handling techniques you will need to use when programming your own applications. Follow up this week's lectures by reading over the associated material in the textbook. Spend at least six hours this week on GUI material.

  The second assignment has been set, so make sure at least that you know what is required. Try to make a start on it this week if you have not done so

already. Spend at least two hours on the assignment.

- Week 17: Monday's lecture completes the material in Chapter 10 on abstract classes and interfaces. Pages 286-299 of the course text cover this. The bulk of study time this week should be spent going over the material covered in the lectures and practising the essentials of abstract classes and interfaces through the class material. Spend at least three hours this week outside class time by coding using abstract classes and interfaces.

  The second assignment was set towards the end of this week. Spend some time looking through it to become familiar with what is expected. It is designed to enable you to practice using abstract classes and interfaces, and add features to existing programs.

- Week 16: Monday's lecture discussed the value of using object-orientation to support simulation, particularly where there are large numbers of objects in the scenario. We covered pages 266-276 of the course text so there is not a lot of background reading to be done as follow up.

  This means that there should be plenty of time to work on the assignment this week. Aim to finish before Friday's deadline because your productivity is likely to decrease during any mad panic as the last few hours tick away.

  The class material encourages you to experiment with the foxes-and-rabbits project.

  In Thursday's lecture we looked at how to refactor the Fox and Rabbit classes to create an Animal class, and the value of defining act as *an abstract method*. This is covered in pages 276-286 of the course text.

- Week 15: Monday's lecture introduced the topics of method overriding, method polymorphism, static and dynamic type. These topics are covered in chapter 9 of the course text, pages 246-256. After the lecture, read over the slides and your notes, and listen to the lecture again if you find that useful. Spend around 4-5 hours this week attending this week's lectures and following up the material covered.

  This week's class material will help you to explore the effects of method overriding for yourself. Be sure to complete the exercises outside class. This should take around two hours in total.

  If you have not already done so, you should certainly start work on the assignment towards the beginning of this week. Spend at least three hours on it this week. The first part is quite straightforward and should get you off to a successful start. Don't forget to read the associated course material to that you are in the best position to understand what is required. If you get

stuck then you are welcome to contact me or your class supervisor.

In Thursday's lecture we completed the material in chapter 9 and then looked at some extra-curricular material on reflection. You should only follow up the material on reflection if you are completely on top of all the standard course material and well on the way to completing the assignment. Please note that this is a *just for fun* exploration and not an official part of the course.

- Week 14: Monday's lecture completed an introduction to the basics of inheritance. Pages 235-245 of the course text support the lecture. Classes start this week, so there is more direction in how to support your study outside of the lectures.

  The class material for this week is more than you will be able to complete within the class, so it is important that you complete this before next week's class. Although class material is not assessed it provides a valuable means to develop your understanding of the material being covered in lectures. Completing it will also mean that you are better prepared to tackle the first assignment.

  Even if you are not planning to start the assignment this week, make sure that you read through it so that you know what you are being asked to do. If you have any questions about what is expected, seek clarification from a member of the course team.

- Week 13: The first lecture set some background to the course as a whole, and started to introduce inheritance. I started with an application initially written using techniques that were covered in CO320, and identified that it would be good to try to reduce some of the duplication inherent in the media classes (CD and Video) and the Database class.

  Chapter 8 of the course text, pages 217-228, support the lecture. Review this material in the text book along with the associated slides. Take some time to read through the source code in the *dome-v1* project. (You can find sources for all of the projects in \\raptor\files\courses\co520\projects.) Make a note in a logbook of anything you don't understand or are not sure about and think of how you could find out more about it. That might involve looking it up in a book, asking a friend, posting an anonymous question or asking the lecturer via email. Make sure you have answers by the end of the week.

  The second lecture continued in Chapter 8 up to the end of Section 8.6 on page 235. Read that far, including having a go at the associated exercises.

  As there is no class this week, you have some extra time to catch up on things you are not sure about from CO320. Make sure that you do spend at least a couple of hours this week actually writing some Java code. That

could involve having a go at some of last term's class exercises that you didn't get around to or to finishing, having another go at one of last term's assessments, or trying an exam question from *last year's CO320 exam*. Another programming idea would be to add another media type to the dome-v1 project to see for yourself how much duplicate code it is necessary to write in this version. That should give you some insights into the value of inheritance when we get to it.

Appendix B: Content according to the module specification

The following learning outcomes and syllabus are taken from the department's approved module specification for CO520.

Subject-specific learning outcomes

Students who successfully complete this module will be able to:

- Use advanced features of an object-oriented programming language, such as inheritance and graphical libraries, to write programs.
- Use object-oriented analysis, design and implementation with a minimum of guidance, to recognise and solve practical programming problems involving inheritance hierarchies.
- Design appropriate interfaces between modular components.
- Evaluate the quality of competing solutions to programming problems.
- Evaluate possible trade-offs between alternative solutions, for instance those involving time and space differences.

Generic learning outcomes

Students who successfully complete this module will be able to:

- Make appropriate choices when faced with trade-offs in alternative designs.
- Recognise and be guided by social, professional and ethical issues and guidelines and the general contexts in which they apply.
- Deploy appropriate theory and practices in their use of methods and tools.
- Make effective use of IT facilities.
- Manage their own learning and time.

The specification also provides the following synopsis.

Synopsis

This module builds on the foundation of object-oriented design and implementation found in CO320 (Introduction to object-oriented programming) to provide a deeper understanding of and facility with object-oriented program design and implementation. More advanced features of object-orientation, such as inheritance, abstract classes, nested classes, graphical-user interfaces (GUIs), exceptions, input-output are covered. These allow an application-level view of design and implementation to be explored. Throughout the course, the quality of application design and the need for a professional approach to software development is emphasised.

The module specification has been deliberately written at a level that is not specific to a particular programming language, such as Java or C++. On the other hand, it is specific to a particular programming style – object-oriented, as opposed to functional or procedural.

Appendix C: CO520 Assignment 1

1 Introduction

This assignment is designed to help you explore the subject of *inheritance* in object-oriented languages. In particular, you will be exploring how shared characteristics of related classes can be represented in a *superclass*, with the specialised elements of the related classes represented in multiple descendent *subclasses*.

The assignment will be marked out of 100 and provisional marks have been associated with the individual components.

You will work with an existing project that does not currently use inheritance, inheritance-assignment.

2 The JUnit Test Classes

The project consists of three classes, HockeyPlayer, Coach and HockeyTeam. In addition, associated with each of these is a JUnit test class. The JUnit test classes test basic functionality, and at least one of the tests fails with the project in its current state. As you develop the project further, you should make use of the test classes and keep them up to date in order to simplify testing and improve the quality of your code.

3 Correcting the printTeam Method (10 marks)

Run the test methods of all of the test classes. If you are not sure how to do this, check the details in chapter 6 of the course text. If you don't have the course text, there is a tutorial on the Unit Testing facilities of BlueJ on the BlueJ web site.

You should find that there is a failure in the testPrintTeam method of TestHockeyTeam. Click on the failure message in the upper window of the Test Results window to obtain details of the failure. The Show Source button will take you to where the error occurred. The problem arises because the class has no coach.

Correct the error by checking for a null coach in printTeam. If there is no coach, print "unassigned". Make sure that team details are still printed even if there is no coach. When you have fixed the problem, rerun all the tests in TestHockeyTeam to ensure that everything is working correctly now.

4 Introducing Inheritance into the Project (30 marks)

The HockeyPlayer and Coach classes share some common attributes - both have a name and a membership id. They also have some common methods - getName and getMembership. Capture these common elements in a new class, Member, that becomes the *superclass* of both HockeyPlayer and Coach.

This change involves placing the common fields and methods into Member and removing them from HockeyPlayer and Coach. Rather than making the changes all in one go, it will be safer to move one field at a time.

4.1 Moving the membership Field

In order to move the membership field you will need to engage in a process called *refactoring*. The aim is to improve the class structures by moving code around, but we are not aiming to introduce new functionality. Once these changes are complete, all the existing tests should still pass.

Start by creating a new class called Member. Modify HockeyPlayer and Coach to indicate that both are subclasses of Member.

Place a membership field in Member and remove the membership field from both HockeyPlayer and Coach. You can do this with cut-and-paste from one of the subclasses if you wish. Make sure that the field has an appropriate comment.

Move the getMembership accessor method to Member and remove the definitions from HockeyPlayer and Coach. Make sure the text of the method comment is appropriate to a shared method.

With membership now a private field of Member, subclasses cannot use the field directly in their methods. Coach must replace its direct access with a call to the public getMembership method it inherits from Member. Make these changes to Coach and check that all classes compile correctly.

Run all of the tests in the test classes to check for any errors introduced during the refactoring process.

4.2 Moving the name Field

When you have successfully moved the membership field to the Member class you can move the name field. The process will be similar.

Start in a similar way as with membership - move the field and getName accessor to Member. Both Coach and HockeyPlayer should be left with a single field each.

Replace direct accesses to the name field in Coach and HockeyPlayer with calls to the inherited getName method. Ensure that everything compiles.

Run all of the tests in the test classes to check for any errors that might have been introduced.

5 Introducing a New Method (15 marks)

It should be possible to change the membership string of players and coaches. Add a method, changeMembership, that makes this possible. Add appropriate tests to check that this method works as it should.

6 Replacing print in HockeyPlayer (15 marks)

We wish to remove printing to the terminal from the HockeyPlayer class.

Replace the print method in HockeyPlayer with a toString method with the following signature:

```
  /**
   * Return a string in the form "Name (goals)".
   * @return A string containing the player's name
   * and number of goals.
   */
  public String toString()
```

The toString method should return a string that is identical with what would have been printed by the print method. For instance, if the player "Alex Forward" has scored five goals, then this method will return "Alex Forward (5)".

In removing the print method from HockeyPlayer, you will have to change the printTeam method of HockeyTeam. The printTeam method should now call a player's toString method and print the string for itself.

Confirm that the existing tests still work, and add further tests to ensure that your toString method works correctly.

7 Introducing Polymorphism (20 marks)

We wish to add a new class to the project, Club:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * Store details of the membership of a Club
 *
 * @author David J. Barnes
 * @version 2006.01.16
 */
public class Club
{
    // The name of the club.
    private String title;
    // The members of the club.
    private List members;

    /**
     * Constructor for objects of class Club.
     * @param The club's title.
     */
    public Club(String title)
    {
        this.title = title;
        members = new ArrayList();
    }
```

```
    /**
     * Add a player to the membership list.
     * @param player The player to be added.
     */
    public void addPlayer(HockeyPlayer player)
    {
        members.add(player);
    }

    /**
     * Add a coach to the membership list.
     * @param coach The coach to be added.
     */
    public void addCoach(Coach coach)
    {
        members.add(coach);
    }

    /**
     * List the membership of the club.
     */
    public void printMembership()
    {
        System.out.println("Membership of the " + title +
                           " Club");
        Iterator it = members.iterator();
        while(it.hasNext()) {
            Object member = it.next();
            System.out.println(member);
        }
    }
}
```

Save a copy of Club in your project folder and ensure that it compiles with your existing classes.

Club was written before common elements of players and coaches were moved to a Member class. It has separate methods for adding player members and coach members to the board. Replace these two methods with a single method called addMember that can take either a HockeyPlayer or a Coach object as its parameter, and adds that member to the membership list.

8 Challenge Exercise (10 marks)

Can you find a way to refactor the project so that it is less tied to *hockey* teams and players and therefore suitable for other types of teams? You may assume that all teams you wish to support have a similar structure of a single coach and a list of players but that some details of players might be a little different. For instance, in basketball teams players don't score goals but points.

Make these changes to your code.

9 Deadline and Submission

The deadline is 23:59 on Friday 3rd February (week 16).

Submit your work via the CO520 submission page. The source files of your project must be stored in a *JAR file* named *club.jar*. A JAR file is easily created from within BlueJ via the *Project* menu.

Select Project/Create Jar File. The *Main class* option can be left as *none* but you must tick the *Include source* box. Select the *Continue* button and save the JAR file with the name *club.jar*.

10 Plagiarism and Duplication of Material

The work you submit must be your own. We will run checks on all submitted work in an effort to identify possible plagiarism, and take disciplinary action against anyone found to have committed plagiarism.

10.1 Some guidelines on avoiding plagiarism:

One of the most common reasons for programming plagiarism is leaving work until the last minute. Avoid this by making sure that you know what you have to do (not necessarily how to do it) as soon as an assessment is set. Then decide what you will need to do in order to complete the assignment. This will typically involve doing some background reading and programming practice. If in doubt about what is required, ask a member of the course team.

Another common reason is working too closely with one or more other students on the course. *Do not* program together with someone else, by which I mean do not work together at a single PC, or side by side, typing in more or less the same code. By all means *discuss* parts of an assignment, but do not thereby end up submitting the same code.

It is not acceptable to submit code that differs only in the comments and variable names that have been used, for instance. It is very easy for us to detect when this has been done.

Never let someone else have a copy of your code, no matter how desperate they are. Always advise someone in this position to seek help from their class supervisor or lecturer. Otherwise they will never properly learn for themselves.

Further advice on plagiarism and collaboration is also available *(local link deleted)*.

You are reminded of the rules about plagiarism that can be found in the Stage I Handbook. These rules apply to programming assignments. We reserve the right to apply checks to programs submitted for assignment in order to guard against plagiarism and to use programs submitted to test and refine our plagiarism detection methods both during the course and in the future.

11 Collection and Retention of Marked Work

Work that has been marked often contains valuable feedback that will help you improve your understanding of the concepts that have been assessed by

an assignment. It is important, therefore, that you collect marked work as soon as it is available. Any coursework that has not been collected by the end of the academic year will be destroyed.