

Redesigning the Intermediate Course in Software Design

C. W. Johnson

Ian Barnes

Department of Computer Science
Institute of Engineering and Information Sciences
The Australian National University,
Canberra, ACT 0200, Australia
Email: cwj@cs.anu.edu.au barnes@cs.anu.edu.au

Abstract

Learning to design software ahead of directly constructing it is a significant hurdle in a Software Engineering education. Our University has run a course in software design for second-year undergraduate students since 1994. We describe the evaluation and improvement of the course as it evolved from 2000 to 2003, from a focus on reverse engineering to forward design, to add design patterns and associated programming tasks, then has redefined its objectives and re-aligned the assessment tasks with them. We evaluated the course in four ways: by the distribution of final grades, subjective evidence on the quality of answers in the final examination, student satisfaction surveys, and comparison of students' final grades with other computing courses taken at the same time. The attempt to improve the course by introducing homework tasks on design patterns did not improve the outcomes. But re-aligning the assessment with the objectives, and introducing a component on requirements specification, improved on most measures.

Keywords: software engineering education, design patterns, software design, course evaluation, alignment of assessment with objectives

1 Introduction: design in the Software Engineering curriculum

Software Design is a clearly identified component in the classic software lifecycle (analysis, design, implementation, testing, deployment, maintenance...). It has a necessary place in a modern software engineering or software development curriculum. But what makes a good course in Software Design? Can it be separated from an embedded practical exercise of project work that covers the whole lifecycle?

The place of software design in the software engineering body of knowledge is secure (Joint Task Force on Computing Curricula: IEEE Computer Society, Association for Computing Machinery 2001), but its positioning in the curriculum of Software Engineering programs is a problem: whether to teach design starting with the introductory courses, in parallel with programming, or leave it until after students have some understanding of the implementation stage, or leave later and combine design with other processes and documentation in a third year

capstone project. The Australian National University is a research-oriented university which offers a four-year degree program in software engineering. The program gave the topic of design a distinct course at intermediate level (second year), ahead of developing and exercising design skills in a further third year course and a third year team software project. This raised the question of just how best to teach design at this level. At this stage in students' learning they are familiar with a constructive approach to creating programs of small and medium-small scale. Design is one of the hurdles because it requires students to articulate expressions of abstract, descriptive views rather than the familiar detailed, operational views.

Teaching design in other engineering disciplines is also seen as difficult (Campbell & Colbeck 1998), but for different reasons: in our own programs we have little difficulty in getting students to be creative and construct their own solutions, but find it hard to have them describe, reuse, and analyse designs.

The university's bachelor degree programs¹ in information technology and software engineering have included a second-year course in software design since 1994. The course is a prerequisite for a third-year teamwork project course which exercises many phases of the software development lifecycle, including the documentation of software design, and for a concurrent third-year course in Software Analysis and Design described recently (Flint, Gardner & Boughton 2004).

Software design has a product and a process component. The product is an abstract structure and architecture, made evident in a collection of documentary descriptions, or in the abstract structure that can be gathered by reading program code. The process is one of discovering or inventing suitable abstractions, creating documents or code, and submitting the design to informal or formal analysis of its qualities. The first of the qualities is correctness of the design to some set of specifications; others are the qualities in the design as a communication between people (coherence, traceability of design elements to requirements etc. (Parnas & Clements 1986), (Meyer 1997)). Aesthetic quality of the design or the resulting code in itself has no place, despite the efforts of early advocates of studying programs for their inherent beauty (Dijkstra 1972), (Bentley 2000), nor does the related quality of usability that can be used to evaluate user interface design. Software design knowledge is well captured and presented in the form of design patterns at this level, although the computing education literature reports attempts to use patterns at the introductory programming level (Clancy & Linn 1999).

Copyright ©2005, Australian Computer Society, Inc. This paper appeared at the Australasian Computing Education Conference 2005, Newcastle, NSW, Australia. Conferences in Research and Practice in Information Technology, Vol. 42. Alison Young and Denise Tollhurst, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

¹We use the administrative terminology of a degree *program* consisting of a number of *courses*.

2 The Software Design course and its evolution

When it was first developed in 1994 the original of this course course included practical work which mimicked the widespread industrial process of reverse engineering. This was intended also to use the study of large existing procedural programs to motivate and exemplify the need for documenting design. The goal of the first version of the course was to address fundamental ideas in the design and construction of programs of nontrivial size, focusing on the notion of design as a process which makes a selection between feasible alternatives, and on design as an outcome of that process (Molinari 1994). This contrasted with other courses that commonly mimic the industrial forward processes of design and construction, as widely used in capstone (typically final year) projects, such as (Adams 1993). Educational mimicry of industrial processes is so widespread as to be almost unquestioned: in a capstone course it is seen as a valuable outcome in itself. But for an intermediate level course we believe that there are more important conceptual shifts to be learnt.

In 2000 the course was enlarged and revised for the enlarged four-year software engineering program. The use of reverse engineering was retained, but the programs used as objects of study and the design descriptions were modified, to include object-oriented programming. Of three assignments the first two were reverse engineering: to attempt to motivate students and refocus the skills from reading and analysis into construction, a third assignment requiring forward design was introduced.

One long-term invariant has been to introduce the course with Parnas and Clements's paper on a rational design process (Parnas & Clements 1986) as a first description of elements and qualities of a design document and an unstructured process for achieving one.

The final examination in 2000 tested mainly descriptive understanding of the processes as described by Parnas, and of a very small vocabulary of design patterns.

In 2001 the course changed lecturers to the current authors, and the point of view of the course changed from reverse engineering and descriptive knowledge of processes, to refocus the actual objectives on the ability to apply the processes of design as well as to describe them. At this stage the formal wording of the objectives were not rewritten. The final examination tested applied knowledge. The failure rates rose sharply (see Appendix 1), but the new objectives were persevered with as they were believed to better prepare students for their third year courses.

Since 2000 the course has been the subject of a process of continual improvement. The desire to improve has been driven by changes in students' preparation (the introduction of object-oriented programming into the introductory curriculum), and changes in the practice of software design (particularly by the encapsulation of design knowledge in the form of software design patterns (Gamma, Helm, Johnson & Vlissides 1995)); and by evaluations of the course. Three kinds of evaluation were used: the lecturers' own subjective judgment, based on reading examination scripts, on how well the learning objectives were being achieved; the results of student surveys requesting categorical ratings or open ended comments about the course and the program; and comparison of student results in this course with other computing courses taken at the same time.

Despite continued surveying and innovations, staff and student dissatisfaction and failure rates remained high. Some of our intended improvements turned out to have little effect. Other changes have improved some evaluation measures with little effect on others. We discuss below the changes that were made in 2002 and 2003, compare the evaluations, and analyse the results.

The changes that were made were to introduce small "homework" tasks in 2002, in an attempt to ground the introduction of abstract design patterns more firmly on concrete program examples. This had little apparent effect. A more substantial realignment of assessment with the course objectives in 2003 gave improved exam performance and a shift in the distribution of student satisfaction, although no substantial improvement in average satisfaction.

3 Attempting improvements: from 2000 to 2002

In 2000 the course was enlarged from its predecessor, and positioned in the fourth semester of the new eight semester software engineering program. (The previous degree programs had been six semesters i.e. three years, and later year courses had been typically sized at twelve courses per year, rather than the new standard of eight to the year). A summary of the course is included at Appendix 2.A1. Over the next two years an introduction to software design patterns was added as the topic became better known and textbooks appeared. The description of the course was revised to re-express the goals, while retaining the use of reverse engineering:

This course is one of a trio: (2.1) Software Construction, (2.2) Software Design, (3.1) Software Analysis and Design. . .

Software Construction takes students from elementary systems of a handful of classes to systems of many tens of classes, large tree structures, suites of processes operating on them, specifications related to systems from the external world real-world, programs bigger than they might chew all in one bite. The **Software Design** course takes the study of software systems up a level of abstraction, not size: abstracting descriptive ideas from source code to its generalised, abstracted description and the patterns of relationships and qualities that apply to this level. The course emphasises the description of designs more than the processes of design. Skills are developed in methods of object-oriented forward design, reverse engineering of design from existing code, and in description and comparative criticism of designs.

(from the original handbook syllabus)

We are concerned here with the design of the course, namely what topics were included, how topics were handled (in lectures, reading, tutorial classes, laboratory classes, laboratory preparatory homework, practical (constructive) assignments, written assignments, mid-semester quiz), and the goals of each assessment within the semester i.e. those that provide learning feedback rather than being for final grading.

The design structure of the course over the period 2000 to 2003 is tabulated in Appendix 2, A1–A4.

3.1 Evaluating the course

Lecturers who present a course have available a range of means for evaluating the design of an individual course. The choice of methods is not always made objectively by educational criteria: some of these instruments are mandated or supported by university educational support structures; some require more or less effort to apply and analyse; some are better suited to evaluating the delivery of a course rather than its design. The evaluation methods that were used on one or more occasions for this course over the period 2000-2003 were:

1. **survey of students**
either (a) a student satisfaction rating, measured as a normative rating on 9 quality aspects, through an in-class, anonymous university-wide survey provided by the university's higher education support centre (CEDAM); or
(b) a course-specific student survey of the course lecturers' own design;
in each case, usually held during the final lecture of the course
2. **grade distribution**
distribution of final grades (tabulated in Appendix 1 for years 2000-2003)
3. **comparable courses**
student cohort performance in other computing courses taken at the same time
4. **examiners' subjective judgment**
judging quality of answers to questions in the final exam
5. **focus group**
analysis of student focus groups discussing the department's programs
6. **retained learning**
open comments from lecturers in subsequent courses, about the apparent knowledge of students from this course
7. **student open-ended comments**
anonymous, open-ended comments appended to the student survey
8. **verbal**
anecdotal open comments from students

We did not intend to run a longitudinal study when we started the process of reforming the course, so not all of these methods were applied in every year. There are clearly more forms of evaluation that might be used but our available time resources were limited.

3.2 Introducing homeworks

In 2002 we introduced a series of homework tasks into the course. These tasks consisted of small reading and programming exercises with design patterns and associated code examples from the textbook (Jézéquel, Train & Mingins 2000). Most students were familiar with the purpose and practice of such homework tasks from their preceding semester course in Software Construction. They were scheduled as preparation for supervised laboratory classes, their satisfactory completion being assessed by eyeball during the lab and explicitly registered by the award of one percentage mark towards the final grade. The course structure was otherwise similar to that in 2001, apart from the addition of a small module on programming with data structures (see Appendix 2.A3).

3.3 Evaluation: 2002

Evaluating the effectiveness and quality of an individual course is hard to separate out from the effectiveness of the whole degree program, the relative popularity of the subject matter of individual courses, and elements of showmanship in their presentation, and whether the course is incremental or paradigm shifting in student understanding.

The course was evaluated by a combination of the methods listed in section 3.1 above: [1(b)] a purpose-designed feedback survey; [2] the grade distribution; [3] direct comparison of grades with a concurrent computing course; [4] lecturers' subjective rating of examination answers against the hoped-for quality of learning; and [5] a larger scope student focus group survey concerning students' first 3 years in the department (performed during the first semester 2003, following this course); and [6] retained learning in a common following course.

The objectives of the course as embodied in the final examination had increased emphasis on design patterns and decreased coverage of design processes. The majority of the questions used in the final examination tested understanding of design patterns, using simplified forward design exercises based on descriptions of situations in the form of informal analysis and requirements for subsystems, and seeking critical comparative commentary such as advantages and disadvantages in applying a particular pattern in this situation.

3.3.1 Examination results and surveys on homework

The practical question was: did introducing design patterns homeworks improve learning? The survey showed that a large proportion of students (50 of 55 responding students) had completed all or nearly all of the homework tasks (5 or 6 out of 6). The survey asked students to rate "how useful were the homeworks in helping you understand the connections between theory and practice in this course on software design?" A large majority (44 of 54 replies) thought that they were 'useful' or 'very useful'; 10 found them 'no use' or 'not much use'. (On average they found the mid-semester quiz slightly more useful, an average rating of 2.28 to 2.09 on a scale of 0 (no use) to 4 ("wouldn't do without it" i.e. essential)).

This appeared to indicate that the introduction of homework had been beneficial. But consideration of the examination results belied this. The exam results in fact were worse than the previous year without homeworks.

	HD	D	Cr	Pass	fail
2001	12	17	20	31	17
2002	9	19	21	21	27

Table 1: Grade distribution 2001-2002 (percentages)

The only significant change was in the proportions in the Pass and Fail groups. The proportions of students in higher grades (Credit, Distinction, High Distinction) were little changed: but the number of failures had increased (from an already high 17% to 27%, at the expense of the relatively large proportion in the Pass grade seen in the previous year (from 31% to 21%). Reinforced by the examiners' subjective feelings, there had been no improvement in understanding as seen in the examination, and this pointed to a decrease in learning among the lower graded students.

What was the cause? was this effect attributable to differences in the course design and content, the textbook, the lecturers' teaching delivery, the student cohort? The textbook was changed (see Appendix 2.A2 and A3) to better match the course content. The lecturers were the same (the authors of this paper), and their enthusiasm undiminished: their (our) engagement with the course is evidenced by the steps made to introduce the homework tasks.

The student cohort was little different. Their university admission scores from secondary school in 1999 and 2000 had the same cutoff. Comparison with the same students' grades in the concurrently offered course in Concurrent Programming supported this conclusion. This course is taken by many of the same students at the same time, and had unchanged content and lecturers in 2001–2002. We compared the average final mark for the two courses. Calculating the average difference in final marks for those students who took both courses (55 of 122 students in 2001, 74 of 127 in 2002) could be expected to show little effect, since in normal conditions the marks in all courses are expected to be close to a norm of 65%. This difference in fact showed a slight, effectively insignificant, improvement for Software Design: from the average trailing by 0.5 percentage points in 2001 to being ahead by 0.8 in 2002. But the distribution of these mark differences for individual students reveals something more significant about students' relative performance in the two courses: in 2002 some 28% of the students taking both courses scored 5 or more percentage points lower in Software Design than in the sister course, an increase in this group from 24% in 2001, while the group who scored 5% or better in Software Design was slightly smaller than before, at 36% in 2001 to 34% in 2002. We take this as evidence that there was a real decrease in performance for students in the Software Design course in 2002 compared to 2001.

3.3.2 Open-ended student comments and focus groups

The open-ended comments by students on their surveys in 2002 were little help in evaluating the course curriculum. The comments tended towards small performance issues: presentation clarity of one lecturer over another, the quality and timeliness of the course notes on the course website. There were few comments on course topics (mainly describing them as “boring” or “uninteresting”, rather than being characterised as irrelevant, or difficult) and quality ratings of textbooks (“book X is useful/useless”) or low-quality, self-contradictory assessments. All comments on the third, forward design assignment were favourable, and several made unfavourable comparisons with the reverse engineering as a difficult task. A couple of comments referred to the amount of memorisation required, perhaps in an attempt to “learn” all of the patterns that were described (though this survey was held before exam study break). A large number of comments referred to the heavy workload for reverse engineering assignments compared to their value in marks.

It appears that students have too little experience (or too much bad experience?) at this stage to provide direct comments on qualities such as alignment of assessment, but one comment is in retrospect an indicator of something going wrong:

The content in Parnas was too advanced for [the] course and had very little in common with what we actually did.

This refers to the seminal paper on design document and process (Parnas & Clements 1986) that we believed to provide a clear and concise description of a design document and a process for its creation, with direct application in the assignment work.

The student focus group that commented on all Computer Science department programs in early 2003 made no useful comments specific to this course, apart from a minor grievance that the same overhead slides were apparently being used in several courses: without further details, we decided that this is probably illustrating the software development roadmap and how various topics corresponded with the development lifecycle.

3.3.3 Missing topics

Another source of evaluation of the course came from lecturers in one of the successor courses run in the first semester of the following year. In particular, the topic of Requirements Analysis was not intended to be covered in this course, but feedback from the lecturer in the following course indicated that students had insufficient knowledge of Software Requirements Specifications, and he argued that learning design was difficult if not impossible without an understanding of requirements.

On reflection we saw that his argument was correct. Although students were exposed to many examples of requirements in the form of their previous programming assignment specifications, these documents were not designed to present a consistent exemplar of how to describe Requirements in the Software Engineering sense, and students had no awareness of these as being “Specifications”. They could be expected to gain some skills in comprehension, but none in application or analytical knowledge as might be expressed in making critical comparisons of the quality of sets of requirements, or (in a more practical direction) improving the faults in a set. Unconsciously we had assumed that students were sufficiently knowledgeable of requirements specifications to understand what lay in front of design. Working from a reverse engineering viewpoint had blinded us to this, in an expectation that the operation of the working software under examination was a form of specification. On examining the curriculum more closely we found that an intermediate level of treatment of the topic had evidently fallen through the cracks between courses, a lesson for the maintainers of curriculum design at the program level.

3.4 Analysis of the evaluation

Despite an in-class multichoice quiz in mid-course, the assignment work was the only component of independent sustained work by students in the course, before their final examination study. The skills developed by the reverse engineering assignments would be hard to examine, and this was not attempted: reading relatively large bodies of code and creating reverse engineering descriptions were not seen as suitable examination matter. The examination concentrated on lecture material, the knowledge of design abstractions expressed in design patterns, which had supposedly been reinforced by the homeworks and labs. But students were being misled by their good assignment marks and good homework results into thinking they understood the material: the final examination was evidently testing something different. The major assessment components—assignments and

examinations—were evidently not aligned with the objectives that we were examining.

In addition, the inclusion of reverse engineering as an explicit objective as well as a means of instruction was decided to be no longer suitable. Although it constitutes a valuable industrial skill it is at odds with student expectations of a design course; research showed that it is rarely if ever included in the curriculum at other universities; and it helped to create an understanding of design abstraction that students were poor at transferring to apply in forward design. The low level code-grounding of patterns was also evidently not transferable to abstract design. Again, the major components of the course were misaligned with the objectives and the final assessment.

Although the third assignment was a forward, creative design it came late in the course. Students had no chance to get feedback on their performance or to consolidate what they had learnt by attempting a further exercise. The transfer between describing an existing program's design by reverse engineering, to creating and describing their own design, was too challenging for all but the more able students. We think that the switch to forward direction also came too late to reduce students' frustration with the course missing their expectations.

4 Retargetting and realignment: the revised course 2003

The analysis of the possible causes led us to redesign the course in four respects (see Appendix 2.A4). We re-stated its goal and objectives; analysed and corrected the alignment of the course objectives with the content and its assessment; enlarged the topic of software requirements; and changed the practical exercises from reverse engineering to one on requirements and two on a single project forward design.

4.1 Course goal and objectives

A major change was to include a substantial amount (three lectures, two one-hour tutorials, one 20% assignment) on the topic of critical understanding of software requirements. Inevitably some other material was reduced to make room, decreasing the number of design patterns described in detail. In response to student comments about the number and size of the assignment work, the other two assignments were modified to become complementary parts of the forward design of a single project: the first part due in mid-semester being worked in pairs of students, requiring a report and presentation to the tutorial group on the high-level design of a solution; the second part due at the end being an individual detailed design for the same problem. Design patterns were referred to this problem where possible throughout their exposition.

The reformulated goal was to develop student skills, from being able to develop software with sole focus on implementation technologies ("program in language X and system Y") to thinking with documentable, conceptual abstractions: namely the requirements (objectives, rather than particular implementation methods); the design ("what" is to be built rather than starting with the "how"); and reusability of software knowledge chunks (software design patterns, versus language libraries).

This goal is defined in more concrete objectives as stated in the revised course description:²

²Such a statement of objectives is part of the ANU Depart-

At the completion of this course students will be able to:

1. use well-structured diagrams and text to describe the design of a medium-scale software system
2. write informal requirements for a medium-small software system
3. create and describe the design of a small scale software system
4. critically compare the design of medium-small software systems for related purposes
5. select and analyse the application of software pattern definitions to a design problem
6. recognise and illustrate the relationships and processes between requirements, design, and implementation in the standard software life cycle
7. demonstrate a reasonable choice of classes and relationships to model system fragments to meet partial system requirements
8. find and select software from that which is openly available to approximately meet system requirements.

The components of the course to achieve these objectives are described as *core content*: students will learn methods for designing software for a given purpose, technical "design ideas" to use (at high level and at detailed level), specifications of requirements for software; and *supporting concepts*: notational methods for describing software design, notations for specification of requirements, software lifecycle framework, and "quality"—what makes it a good design (or not).

As an example of the extent of the redesign: the lectures on Parnas and Clements's paper describing a rational design process (Parnas & Clements 1986) were moved from the introduction to the conclusion of the course (now presented as "this is a way to encapsulate and compare what you now know how to do" rather than "this is an advance description of how you can do something that you have not yet encountered").

At the same time a new textbook became available and was selected as it appeared to be a better match to the course (Braude 2004).

4.2 Alignment of learning and assessment

The revised course aligns assessment with learning by means of:

- **software requirements topic**
 - three lectures describing qualities of a good set of specification statements, and two one-hour tutorial classes exposing sets of examples to critical analysis and creative improvement. The example provided was a multiple-alarm clock, whose functions could be demonstrated by a black-box simulation program.

ment of Computer Science standard documentation of courses. It is interesting to observe how the explicit documentation of course objectives is gradually increasing the degree of "assessability" or performance-related competence in their description as current curriculum improvements for accountability are applied, which has also helped to expose the misalignment of objectives and assessment.

- a first assignment that requires critical analysis and improvement of a set of requirements, and creation of a small set from a narrative description. The example chosen was the simplified functions of a controller and wall panel for domestic thermostatic heater and cooler devices.

This assessment contributes to objectives: 2, 6.

- **high level design/software architecture topic**

- a second assignment requires a short written report and a presentation, working in pairs, to produce a high level design for the project that is also the subject of the third assignment. The assignment specification is a set of requirements statements of the same kind as in the first assignment, and part of the assignment deliverable was a correction and refinement of these requirements. This form of assignment neatly integrated and reinforced these aspects from the first part of the course, allowing a second pass at this process of criticise and improvement in a more strongly motivating setting.

This assessment contributes to objectives: 1, 2, 3, 6.

- **detailed design: documenting, UML, patterns**

- the third major assignment requires integration of design description notations, selection of designs from UML, comparison with existing implementation programs

This assessment contributes to objectives: 1, 3, 4, 5, 6, 7.

- **assessment and examination changes**

- pairwork, presentation and feedback
The second assignment was chosen to require work in pairs and to deliver a presentation to a small audience (the tutorial class) as well as a short written design report. This was designed to contribute to building students' verbal communication skills, and to provide feedback instantly from an assessor rather than having it delayed by the several weeks of normal assignment marking and a spanned short vacation break.
- assignment workload
By combining two assignments on one project it was intended to decrease the background familiarisation time and increase confidence that they understood the assignment and its objectives well ahead of the due date, and to provide feedback on the preliminary design well ahead of the final.
- open book exam
in response to the comment about memorising details, for the first time in our own experience we provided an open book exam: students were allowed to bring in the text or recommended book and one page of notes. Our hidden motivation is the hope that by causing students to condense the important items to a page of notes, and to learn their way around the important parts of the

textbook, they are being induced to structure their study, organise their learning, and make notes in a productive way.

4.3 Evaluation and analysis

The evaluation available after the course consisted of examiners' subjective impressions from the exam questions; distribution of final exam grades; a university-standard student satisfaction survey taken before the end of the lectures; and verbal open comments from students; and an anecdotal report from a third year course in the following semester.

4.3.1 Examiners' impressions

4.3.2 Grade results and course comparison

As the tabulated distribution of final grades show (Appendix 1) the total failure rate dropped back to the 2001 level (18%), while the number of Credit and Distinction grades increased (to 28% and 21%) at the expense of the Pass grades. This is seen as an improved result, although the absolute level of the failure rate is still a concern. What is more, there was no need for the 5% upward scaling adjustment of exam marks deemed necessary in 2002.

The comparator course was affected by a change in staffing and format, so the comparison may not be as valid as that in 2001 and 2002. The difference in average marks between it and Software Design grew from 0.8 to 3.6 in Software Design's favour.

4.3.3 Student surveys

The university's standard student survey provides normative ratings over 9 aspects of a course: workload, organisation, teaching and learning methods, quality of support materials, availability of staff, intellectual challenge, assessment for grading, feedback for learning, overall impact on development. The first look at the result was a disappointment, where we had expected a more successful evaluation: of 64 students surveyed, the average student satisfaction with the four aspects we expected to improve most (course organisation, methods, assessment, and feedback) was a rating of 3.75 out of 7 (described as Borderline-Satisfactory). But this average was lowered by a small number (8-16% of students) of very low ratings in the Unsatisfactory/Very Poor class; more encouraging is the similar number in the Very Good/Excellent class, not previously seen. The proportion rating these aspects as Good or better (Good, Very Good, Excellent) is 28% (organisation), 33% (teaching and learning methods), 27% (assessment), 33% (feedback—with a further 35% Satisfactory). The intellectual challenge was rated Good or better by 51%, the workload by 66%.

4.3.4 Subsequent courses

Anecdotal reports from the lecturer of the successor course in Software Analysis and Design run in 2004 are that students have a much better understanding of requirements, as intended.

5 Discussion

Several changes were introduced at the one time, so can their effects be untangled? In all years the student survey was carried out before the exam, so that

students' experience of the change in exam format to open book would have no effect at that stage.

As examiners, we found a much better understanding of requirements and design and their relationship in exam answers: the subjective understanding of similar patterns/situation questions was better than previous years, despite less time spent in lectures on design patterns. The combination of changes evidently succeeded in improved outcomes, to some extent.

5.1 Open comments from students

No written comments were collected in 2003, but there was one notable verbal comment: "why didn't you tell us about Parnas at the beginning? it would have been so useful", after previous student comments had indicated to us the need to move the lecture on the Parnas and Clements paper from the introductory module in 2002 to the final module in 2003. It is a reminder that there is no one right order of material for everyone.

5.2 Towards inquiry learning

Although the initial design had higher intentions, we made only half a step towards making a design course around inquiry-based, student-contributory learning. In the early stages of redesign we explored creating an inquiry-based, student-contributory course, as advocated in one of the current educational reform movements (Smith & Waller 1997). Our fear is that the goal of "learning to design" as a motivator is not strong enough to lead all students through the challenging process of learning to abstract and document designs, against the seductive, addictive pull of "just programming it".

We observe that a course in Software Design is bound to frustrate many expectations. The student interested in programming who wants to "just learn how to create bigger/smarter programs" will be frustrated, because there is no programming implementation. The student interested in wider-scoped information systems who wants to extend Analysis into Implementation via Design will also be frustrated because there is no development of Analysis, though there is some attention paid to its product, namely Requirements. The student who expects to learn about designing human user interfaces will be disappointed: although the word "design" appears, students have not previously come to face the distinction between interface design and internal design. And the student who sees the getting of software knowledge as the amassing of programming language knowledge and cool trick effects will be frustrated.

These expectations may be unjustified, but in designing a course we are setting ourselves up to frustrate all kinds of expectations in the name of better quality software engineering, quality software development, and higher education skills of knowledge awareness and higher descriptive and critical thinking. Hence there is a need for a strong motivating activity to sweeten the bitter pill, and to bump both students and teachers out of the rut of expectation.

The means of motivating students changes from year to year with the volatility of the job market and the changes in student initial motivations in entering university computing courses. Anecdotally, our computing students are keen on the constructive aspects of their courses, and more reluctant to engage with theory, analysis, testing and design documentation—perhaps because these topics often use manageably

small examples which are quite a long remove from everyday constructive programming. The capstone project course at the end of the program has been seen as a corrective, being a large enough program tackled by sufficiently mature students to appear to be more realistic.

A very early description by Freeman of the choice of projects needing to have a motivating and grounding aspect, producing a system that is "useful to users other than the students" (Freeman 1976), is echoed in Ben Shneiderman's manifesto over 20 years later in his book *Leonardo's Laptop* (Shneiderman 2002). Shneiderman proposes restructuring computing education in a student learning centered approach, containing elements identified as "*collect, relate, create, donate*". The structure is a form of inquiry learning, with small team based project work that has the target of donating something of value to the community. This can be applied at all levels, not only the capstone.

One of us (Johnson) intends to provide such a motivator in the major assignment component of the software design course in 2004. While it is unlikely to be a strong enough motivator to make us confident about changing to a complete inquiry learning approach, the idea is attractive. The Open Source movement is seen as one of the main means for "donating" software to one community, at least. While documented forward design is not a noticeable feature (and is often scorned) in this movement, the larger open source user community is represented by not-for-profit organisations for example, and those with environmental concerns. These areas will be examined for possible motivating design projects that have the possibility of students following up with their own implementations, donating them for the public good.

5.3 Conclusion

This is a difficult course for students, and for teachers. In the first three semesters of our program it is possible for students to pass by demonstrating practical, constructive programming skills. The more abstract and theoretical material included in these courses can be avoided, if the student is willing to scrape through. The software design course in their fourth semester is the first time they are forced to engage with abstraction, and this comes a shock to many students who deny, resist and fail, or struggle.

The history of our design of the course is the history of our coming to terms with this conceptual and educational gap. The gap is a source of pain for students: but it is also a source of difficulties for teachers to identify what is so different about the Design course in among the technology and theory courses. As pointed out by Keith Devlin in a short piece on the role of mathematics in software engineering education,

... software engineering is all about abstraction. Every single concept, construct, and method is entirely abstract. Of course, it doesn't feel this way to most software engineers. But... the main benefit they got from the mathematics they learned in academia was the experience of rigorous reasoning with purely abstract objects and structures. Moreover, mathematics was the only subject that gave them that experience. It's not what was taught in the mathematics class that was important; it's the fact that it was mathematical. (Devlin 2001).

In redesigning the Software Design course twice we came to recognise that the real goal is to create a bridge to an abstractive, descriptive view of software, from what has previously been constructive and creative. If this is the goal, it has to be realised in the course objectives. We do not want to rely on an osmotic transfer of learning about abstraction from students' small exposure in mathematical courses to their software engineering courses: we see a real need to make explicit the step into computing-specific non-mathematical abstraction.

With our second attempt to improve this course, we seem to be on the right track. The changes that worked grew out of reconsidering the objectives of the course, including looking at those courses that precede and that follow it. We then modified the material, the order of delivery, and particularly the assessment, so as to align them with the revised objectives, as listed in section 4.1. This more comprehensive approach was more successful than a rather piecemeal attempt the previous year.

There is plenty more to investigate here. One interesting possibility going to the heart of the subject of this course is to take into account the effect of students' personality type. In the Myers-Briggs framework, (Myers & Myers 1980, Keirse 1998) two of the contrasting types are Sensates (S) and Intuitives (N). The detail focus needed for programming comes naturally to Sensates, while the ability to see the big picture and to move between levels of abstraction—needed for design—come more naturally to Intuitives. At the moment we don't know anything about the balance of personality types in our class, but this raises several questions.

Further evaluation of the students is also desirable to provide a better foundation for our conclusions. Systematically following up students' results in later courses, and after graduation, would provide further useful data, but (alas) tracing and getting access to the students with feasible effort and resources has made this too difficult to do.

The most general conclusion is to do with the *real* goals of computing courses. Often a significant part of what we really want students to learn has little to do with the material covered or the published course objectives. In many disciplines many of the different expectations are unarticulated under the labels distinguishing “introductory”, “intermediate” or “advanced” courses: as we have alluded to in the title of this paper. In COMP2110 Software Design, an important goal is for students to learn “abstraction”, but this isn't stated explicitly anywhere. This is similar to the common student misunderstanding about the purpose of the mathematics prerequisites for many computing courses. We don't really care if they know about differential equations or generating functions. What we want is something much harder to nail down, perhaps “mathematical maturity”, or “the ability to reason within abstract systems”. Perhaps there is significant improvement to be found by digging deeper, below the stated objectives, and then realigning our courses—or indeed our whole degree programs—to address these real goals.

Acknowledgments

We acknowledge the very large contribution made by the original course managing lecturer Brian Molinari (now Director of Scholarly Information Services, Division of Information, ANU in the original conception, design and realisation of the Software Design course and its integration with the Software Engineering pro-

gram curriculum. The need to redesign the course ten years later is no criticism of the original design.

Chris Johnson is also grateful for the intellectual stimulation in discussions and support given by Mandy Lupton (now of Griffith Institute of Higher Education, Brisbane) and the members of the 2003 ANU Centre for Development of Academic Methods course in Curriculum Design and Innovation, who offered insights from education in many areas from Anthropology through Music to Zoology, into what to them is our rather peculiar discipline of computing.

References

- Adams, E. J. (1993), ‘A project-intensive software design course’, *ACM SIGCSE Bulletin* **25**(1). Proceedings of the 24th SIGCSE technical symposium on Computer science education.
- Bentley, J. (2000), *Programming Pearls*, 2nd edn, Addison-Wesley. ISBN 0-201-65788-0.
- Braude, E. (2001), *Software Engineering: an object-oriented perspective*, Wiley. ISBN 0-471-32208-3.
- Braude, E. (2004), *Software Design: from programming to architecture*, international edn, Wiley. ISBN 0-471-42920-1.
- Budgen, D. (1994), *Software Design*, Addison-Wesley.
- Campbell, S. & Colbeck, C. L. (1998), Teaching and assessing engineering design: A review of the research, in ‘ASEE Annual Conference Proceedings’, American Society for Engineering Education *check*.
- Clancy, M. J. & Linn, M. C. (1999), ‘Patterns and pedagogy’, *ACM SIGCSE Bulletin* **31**(1). The proceedings of the 30th SIGCSE technical symposium on Computer science education.
- Devlin, K. (2001), ‘Viewpoint: the real reason why software engineers need math’, *Communications of the ACM* **44**(10), 21–22.
- Dijkstra, E. W. (1972), ‘The humble programmer’, *Communications of ACM* **15**(10), 859–66.
- Flint, S., Gardner, H. & Boughton, C. (2004), Executable/Translatable UML in computing education, Vol. 26(5) of *Australasian Computer Science Communications*, Australian Computer Society, CRPIT vol. 30, pp. 69–75.
- Freeman, P. (1976), Realism, style, and design: Packing it into a constrained course, in ‘The papers of the ACM SIGCSE-SIGCUE technical symposium on Computer science and education’.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley.
- Jézéquel, J.-M., Train, M. & Mingins, C. (2000), *Design Patterns and Contracts*, Addison-Wesley.
- Joint Task Force on Computing Curricula: IEEE Computer Society, Association for Computing Machinery (2001), *Computing Curricula 2001 Computer Science*, Technical report, IEEE Computer Society and Association for Computing Machinery.

Keirse, D. (1998), *Please Understand Me II: Temperament, Character, Intelligence*, Prometheus Nemesis Books. ISBN: 1885705026.

Meyer, B. (1997), *Object-Oriented Software Construction*, 2nd edn, Prentice-Hall. ISBN 0-13-629155-4.

Molinari, B. P. (1994), Unit description for Computer Science COMP2038 Design of program systems. version 1.6 Australian National University Department of Computer Science unit database, 1st series.

Myers, I. B. & Myers, P. B. (1980), *Gifts Differing: Understanding Personality Type*, Davies-Black Publishing. ISBN 0-89106-074-X.

Parnas, D. L. & Clements, P. C. (1986), 'A rational design process: how and why to fake it', *IEEE Transactions on Software Engineering* SE-12(2), 251–257.

Shalloway, A. & Trott, J. (2002), *Design Pattern Explained: A New Perspective on Object-Oriented Design*, Addison Wesley.

Shneiderman, B. (2002), *Leonardo's Laptop: Human Needs and the New Computing Technologies*, MIT Press. ISBN 0-262-19476-7.

Smith, K. A. & Waller, A. A. (1997), Afterword: New paradigms for engineering education, in W. E. Campbell & K. A. Smith, eds, 'New paradigms for college teaching', Interaction Book Company, Edina, Minnesota, USA.

Appendix 1: Final grade distribution 2000-2003

	HD	D	Cr	Pass	fail
2000	8	27	39	16	9
2001	12	17	20	31	17
2002	9	19	21	21	27
2003	7	21	28	24	18

Table A1: Grade distribution 2000-2003 (percentages)

A different lecturer taught and examined the course in 2000. The authors taught and examined 2001-2003.

Appendix 2: Design of COMP2110 Software Design 2000-2003

The number of lectures devoted to each topic is an approximate indicative of the weighting within the course.

A1. Year 2000

Objectives

- reverse engineer a high-level design, given a moderate sized program in the programming languages Eiffel, Java and C,
- describe a range of important design architectures,
- develop, prototype and record a detailed design, given a high level design document, and
- develop and record an initial high-level design, given a specification document.

Textbook (Jézéquel et al. 2000)

Recommended books (Meyer 1997), (Budgen 1994)

Assessment

topic	assessment weight
1. trivial LaTeX familiarisation	2.5%
2. Webtext system I reverse-partial	7.5%
3. Webtext II reverse and modify	15%
4. ASEAM system reverse	15%
5. improve ASEAM system	20%
Homework and tutorial contribution	n/a%
Quiz	n/a
Final exam	40%

A2. Year 2001

Objectives

(no formal change from 2000)

Textbook (Jézéquel et al. 2000)

Recommended books (Meyer 1997), (Budgen 1994)

Topics & Modules: lectures, tutorials, laboratories

module	lectures	tuts	labs	home-work
Process	2(intro)	-	3	-
Architecture, quality	4	-	-	-
Documenting	3	2	3	-
UML, patterns	12	-	2	-
Other: admin, intro to assignments(4), review	5	1	-	-

Assessment

topic	assessment weight
1. Webview–reverse engineer	12.5%
2. Armidale system–reverse engineer	17.5%
3. simulation of passenger flow in railway system–forward design	20%
Homework and tutorial contribution	n/a
Quiz	10%
Final exam	40%

A3. Year 2002

Objectives

(no formal change from 2000)

Textbook (Shalloway & Trott 2002)

Recommended books (Gamma et al. 1995), (Jézéquel et al. 2000)

Topics & Modules: lectures, tutorials, laboratories

module	lectures	tuts	labs	home-work
Process	2(intro)	-	3	-
Architecture, quality	4	-	-	-
Documenting	3	1	3	1
(Data structures*)	4	-	1	
UML, patterns	9	-	2	5
Other: admin, intro to assignments(4), review	5	-	-	-

**additional ring-in topic*

Assessment

topic	assessment weight
1. Webview–reverse engineer	12.5%
2. Armidale system–reverse engineer	17.5%
3. passenger lift simulation forward design	20%
Homework and tutorial contribution	5%
Quiz	10%
Final exam	35%

A4. Year 2003

Objectives

1. use well-structured diagrams and text to describe the design of a medium-scale software system
2. write informal requirements for a medium-small software system
3. create and describe the design of a small scale software system
4. critically compare the design of medium-small software systems for related purposes
5. select and analyse the application of software pattern definitions to a design problem
6. recognise and illustrate the relationships and processes between requirements, design, and implementation in the standard software life cycle
7. demonstrate a reasonable choice of classes and relationships to model system fragments to meet partial system requirements
8. find and select software from that which is openly available to approximately meet system requirements.

Textbook (Braude 2004)

Recommended books (Gamma et al. 1995), (Jézéquel et al. 2000), (Braude 2001), (Shalloway & Trott 2002)

Topics & Modules: lectures, tutorials, laboratories

module	lectures	tuts	labs	home-work
Process	2(intro) 1(how-to) 1(final)	1	-	-
Requirements	3	2	-	-
Architecture, quality	3	-	-	-
Documenting	3	1	3	1
UML, patterns	6	-	2	2
Other: admin, Q&A(2), review	5	-	-	-

Assessment

topic	assessment weight
1. Requirements: critique and create (alarm clock, thermostat)	15%
2. High level design report + presentation (personal calendar)	10%
3. detailed design report (personal calendar)	30%
Homework and tutorial contribution	5%
Quiz (none)	-
Final exam	40%