

Teaching Patterns and Software Design

Ian Warren

Department of Computer Science
University of Auckland
Auckland
New Zealand
Email: ian-w@cs.auckland.ac.nz

Abstract

In this paper we describe our experiences with reengineering an undergraduate course in software design. The course's learning outcomes require that students can model, design and implement software. These are inherently practical skills and rely on *functioning* knowledge. To facilitate a learning environment in which students can acquire the necessary deep level of understanding, we have designed the course by applying the educational theory of *constructive alignment* and a number of proven techniques for teaching, learning, and assessment. Fundamentally, we have embraced the *active learning* paradigm that recognises that student activity is critical to the learning process. In this paper, we describe several active learning techniques that we have used including role play, problem solving and peer learning. We also describe two novel assessment techniques we have developed, *holistic assessment* and *formative examination*. In addition we describe how students work with JUnit, a popular unit testing tool, not as users but as developers by applying design patterns to extend it with new functionality. Finally, we report on student assessment results and relay student feedback.

Keywords: Active and peer learning, formative and holistic assessment, constructive alignment, design patterns, JUnit, UML.

1 Introduction

Prior to the expansion of tertiary education, the traditional model of teaching was, for the most part, effective. With this model, teachers typically expounded their knowledge to students in the form of lectures. Students learned passively, listening to what the expert had to say, and were typically assessed exclusively by the unseen exam paper. This model worked well for many years principally because the students admitted were among the most academically gifted of their generation. Consequently they required no stimulus to use their high-level cognitive processes and so teaching these students involved relatively little effort. A failed student indicated a weak student, and, in general, did not cause lecturers to reflect on their "teaching".

Today's tertiary climate is very different. An explosion in student enrolments has led to significant

increases in class sizes and much diversity in the student population. Today, it is common for a class to include students with multicultural backgrounds from native and foreign countries. The majority of students are typically continuing uninterrupted through their education alongside a smaller cohort of mature students. These factors result in students with a much richer cross section of academic ability than their earlier counterparts. The result of such diversity is that traditional approaches to teaching, learning and assessment in tertiary education need rethinking. A variety of techniques should be employed since one size does not fit all (Biggs 1999, Bligh 1971, Honey & Mumford 1995, McKeachie, Pintrich, Lin & Smith 1986, Race 1993). In addition, the range of techniques should include ways of engaging less academically-oriented students in their learning.

Effective teaching requires two bodies of knowledge: *content* and *teaching*. The former is subject matter expertise and is clearly a prerequisite for quality teaching. The latter is knowledge of how best the content knowledge can be taught and learned. Today, content knowledge alone is insufficient to facilitate an environment for learning which caters for student diversity.

While universities have been used to research assessment exercises such as the RAE (UK) and the PBRF (New Zealand), they are now invariably subject to teaching quality audits too. To ensure effective teaching, universities generally offer programmes to teach their teachers how to teach. At Lancaster University, for example, new academics are required to attend the CiLTHE (Certificate in Learning and Teaching in Higher Education) programme. Reflecting on one's teaching is fundamental to the programme and this drives improvement. The programme involves participating in peer observation sessions (as subject and observer), reviewing research concerned with *teaching* knowledge, and participating in workshops that cover basic aspects of teaching and learning. Completing the course requires writing a dissertation that integrates these activities with particular emphasis on reflection and measures taken to improve teaching and learning effectiveness.

This author was educated prior to the mass expansion of the tertiary sector and began his teaching career using the traditional teaching and learning techniques that he experienced as an undergraduate. Participating in the CiLTHE programme triggered reflection on these practices and made us aware of a general paradigm for learning known as active learning, reviewed by Warren (2002). The basic philosophy of this paradigm is to engage students in their learning by having them *doing* activities rather than being passive recipients of knowledge. Active learning engages students and provides the stimulus for

students to use their higher-level cognitive processes that might otherwise remain dormant. Furthermore, active learning fosters deep understanding of subject matter.

In this paper we present our experiences with applying the active learning paradigm to teaching an undergraduate course in software design. A particular learning outcome of the course is that students should be able to *do* design by making effective use of design patterns.

There is an interesting parallel between teaching and learning on the one hand and design patterns on the other. Design patterns are little more than good practice; they are the culmination of tried and tested techniques for designing software that exhibits desirable properties like flexibility and reuse (Gamma, Helm, Johnson & Vlissides 1995). On several occasions, undoubtedly along with other software designers, we have solved a design problem only to later find that what we have actually done has been to apply a particular design pattern. This is reassuring, but the point is that we have subconsciously applied what is accepted to be good practice. With teaching, we often do the same; we unknowingly use a technique that has its roots in established education theory. In both cases, however, adopting proven techniques is important to yield quality results.

In Section 2, we review a subset of relevant education theory, identifying labels for good practice, and elaborate on these. We continue in Section 3 by introducing the software design course that we have reengineered. In particular, we specify its learning outcomes for students and discuss the key decisions we made to facilitate students satisfying these outcomes. In Section 4, we discuss some concrete applications of active learning in the context of our course. In particular, we include experiences of role play, problem-based learning, and peer-learning. In Section 5, we detail two novel assessment techniques that we have developed, holistic assessment and formative examination. We continue in Section 6 with a discussion of students' results and feedback. Finally in Section 7, we close with some concluding remarks.

2 Teaching and learning principles

Education research is a substantial and active area of work. In this section, we restrict our discussion of sound teaching and learning practice to focus on the fundamental idea of constructive alignment, relevant aspects of teaching and learning styles, and assessment. With this subset of theory in place, contributing to our teaching knowledge, we can make decisions about how best to teach the subject matter of software design.

2.1 Constructive alignment

Constructive alignment (Biggs 1999) is the process of synchronising teaching methods, learning activities, and assessment tasks with a course's learning objectives. Alignment of each of these three elements with learning outcomes is crucial for effective teaching. Clearly, what a student does should be informed by course objectives, and the way in which they are assessed should test the extent to which they have met course objectives. Also, any teaching activities should be driven by course objectives and should support students in their learning activities and prepare them for assessment.

Constructive alignment clearly requires that the learning objectives be identified and made explicit in order to align elements with them. This itself is no more than good practice. For teachers, it guides their decision making with respect to developing the teaching, learning and assessment elements of a course. For students, transparent criteria informs them of expectations and enables them to assess themselves.

Biggs suggests that course objectives should be defined by considering the level of understanding required of students. Bloom & Krathwohl (1956) define a fine-grained taxonomy for different levels of cognitive activity which we have abstracted into *declarative* and *functioning* knowledge. The former represents an ability to recall facts and definitions while the latter is concerned with problem solving skills, assessing alternatives, and generally informed decision-making. For example, an introductory Java programming course might require that students develop a deep understanding of basic programming constructs, objects and classes, necessary for writing their own solutions to problems. Conversely, students might also be required to come away with an awareness of the JDK's commonly used packages, such as `awt`, `util` and `applet`. Programming requires *functioning* knowledge that enables students to analyse a problem, develop a solution and make appropriate use of programming constructs. Awareness of Java packages requires only *declarative* knowledge.

In the above example, learning activities require students to be active in order to build the required depth of understanding for programming. Appropriate activities include problem-based-learning where students engage in writing programs to solve problems. Aligned assessment in this case would again involve students writing code to demonstrate that they have developed the performative level of understanding sought. To assess students' awareness of packages, a teaching activity that simply introduced key packages would be sufficient. An appropriate assessment in this case would be a written test, where students simply demonstrate that they can recall the name of a package for a particular purpose.

2.2 Teaching and learning styles

Education theorists have developed models, summarised by Blair (2000), that attempt to characterise how students learn. To illustrate a representative model, we draw on the Ripple Effect model (Race 1993) which comprises four concentric layers. The act of *wanting* is at the core of this model and addresses motivation; when motivated a student is interested, inquisitive and has an intrinsic desire to learn. A motivated student will put theory into practice and learn through trial and error (*doing*) in layer two. Progressing outwards to the third layer, *digesting*, a student reflects on his or her activity from layer two in the context of the course and beyond. The outermost layer, *feedback* denotes others' reactions to a student's work and includes feedback from assessment.

Honey & Mumford (1995) define a similar model which has in common with Race's recognition that learning *is* the process of student activity. Only from actively engaging in such activity do students build and evolve complex semantic structures which equip them with functioning knowledge. This fuels the argument for a teaching style that facilitates active learning.

The traditional lecture is one technique available to teachers to conduct teaching sessions. As Biggs

Most people learn:

- 10% of what they read.
 - 20% of what they hear.
 - 30% of what they see.
 - 50% of what they see and hear.
 - 70% of what they talk over with others.
 - 80% of what they use and do in real life.
 - 90% of what they teach someone else.
-

Table 1: Effectiveness of different sensory modalities

(1999) points out, exclusive use of traditional lecturing is questionable. Lectures do not stimulate high-order thinking since students remain passive recipients of information. Table 1 (Biggs 1999) shows that the passive activities of hearing and seeing are relatively ineffective forms of learning. Lectures can be effective, but no more so than alternative techniques. In addition, lectures exceed students' concentration span by a factor of 3 for a 50 minute lecture and the pace of delivery will not suit all students. This is a particular issue today because of student diversity. Finally, there is little scope for feedback on the extent to which students are absorbing material during a lecture.

Simple techniques for improving the effectiveness of lectures include breaking the monologue by throwing open a question to students. To work, time must be given for students to think, employing their high-level cognitive processes, to address the question. In other cases, a change in activity, such as watching a video clip or engaging in a short task, is effective at restoring concentration levels close to that of the first 15 minutes of the lecture (Bligh 1971).

More radical departures from traditional lecturing include peer-learning. As Table 1 shows, the activities that have a significant and positive impact on learning include talking things through with others, and teaching others. In these cases students are active in their learning. Peer-learning can be integrated with lectures where students work in pairs or small groups on a short task. Of course, peer-learning is not restricted to lectures. As we describe later, peer learning can be used in lab sessions and directed study outside of scheduled sessions.

2.3 Assessment

A *valid* assessment task is one that is aligned with course objectives (Biggs 1999). As Boud (1995) points out, assessment is the most influential single factor governing what and how students learn. Boud goes so far as to say that assessment can often have more impact on learning than teaching materials. We concur with this since, as we have described above, it is student activity that determines what a student actually learns. Assessment is thus important to get right.

Assessment tasks can be classified in two groups: formative and summative. Historically, summative assessment dominated tertiary education, being manifested by the traditional exam. Summative assessment has its place in grading students, but it offers minimal feedback for students since all that is returned to a student is a mark. There is typically no explanation for the mark, neither are there any comments on what was done well, what was done poorly,

and what was misunderstood. Formative assessment is now commonplace in Universities, often taking the form of coursework. The point of formative assessment is to provide feedback to both students and teachers. Ideally, formative assessment is returned to students when it has most value, in preparing for exams. However, it is unfortunate that in today's environment of large classes and competing pressures on academics' time, coursework is often returned late, with limited feedback, and consequently of limited value.

The traditional unseen exam remains ubiquitous because it is a *rigorous* means of assessment; there is no question that it is a particular student's own work that is being assessed. A key criticism of the traditional exam, however, is that it tests students under artificial conditions, being time constrained and relying exclusively on memory. Common variations on the traditional exam include open book exams, which relieve the burden on memory and allow for more problem-oriented questions, and seen papers which allow students to focus on preparation. However, both variations may compromise rigor.

Any assessment task should also be *reliable* (Biggs 1999). A task is reliable if two conditions hold true: first, one assessor would make the same judgement about the same piece of work on different occasions; and two, different assessors would make the same judgement about the same piece of work. Reliability is thus about fairness. Furthermore, there is a clear relationship between reliability and constructive alignment. The probability of an assessment task being reliable is higher where the learning objectives are transparent and explicit since assessors know what they should be assessing.

3 Course overview

The Software Design course that we developed forms part of the Software Engineering stream of the Computer Science degree at Lancaster University. The course is taken by all second year Computer Science undergraduates.

3.1 Learning outcomes

In designing the course, we identified the learning outcomes up front since they guide subsequent decisions governing teaching, learning and assessment activities. At the end of the course, we expect that students are able to:

1. *Identify and describe the objectives of software design.* Design objectives include correctness, robustness, flexibility, reusability and efficiency (Braude 2003). Students should appreciate that software should not only be correct, but that the latter 4 non-functional objectives are also important.
2. *Interpret and construct UML models of software.* The UML, being a standard industry notation, is an obvious choice for communicating design knowledge. Essentially, students should be able to read and write UML models.
3. *Explain the notion of design patterns and describe a subset of patterns.* Design patterns embody proven design solutions (Gamma, Helm, Johnson & Vlissides 1995). Students should appreciate that using patterns fosters an engineering approach as opposed to one that solves problems from first principles.

4. *Apply patterns to solve real-world problems, making sensible tradeoffs where necessary.* Awareness of patterns is important but is no substitute for being able to apply a pattern to solve a problem. This objective is concerned with deeper, functioning knowledge.
 5. *Apply newly acquired and developed programming skills.* Students have functioning knowledge of fundamental OOP but have limited knowledge of Java class libraries and more advanced aspects of programming. This objective aims to equip students with stronger implementation skills.
 6. *Work with relatively large software projects.* Similarly, students' experience of developing software is typically confined to small programs involving a few classes. Exposing students to larger projects is good preparation for final year project work and industrial practice.
- *Facilitate active and peer learning.* Active learning is essential for effective learning and acquiring functioning knowledge necessary to satisfy objectives 2, 4, 5 and 6. Similarly peer learning offers a means for students to add value to their learning experience.
 - *Use formative assessment tasks.* Formative tasks provide valuable feedback to students. However, we are interested in using feedback mechanisms that generate feedback at a time when it remains beneficial to students and at low cost to staff running the course.
 - *Adopt a real software product for students to work on.* We feel that using a software product that has an established user base is motivating for students as it is a realistic piece of software. Furthermore, doing so reduces course preparation time since the amount of software to be written for the course is reduced.

The decision to introduce design patterns and to use the UML requires little justification. Design patterns have attracted much interest from industry and academia alike in recent years and we feel that they are an essential tool for Software Engineering graduates. Similarly, the UML is a natural complement to design patterns, providing a means for students and teachers to communicate them. An ability to cope with larger projects is also important since this introduces more realism with a need to work with others, to decompose the solution space, and to organise development artefacts such as source code, executable code, test code and design documents. Furthermore, a relatively large project offers an effective way for students to see the value of using design patterns and how individual patterns can be synthesised to solve a larger problem.

The learning outcomes involve a mix in the level of understanding sought from students. Outcomes 1 and 3 seek only declarative knowledge. The verbs 'identify' and 'describe' used in the vocabulary of these objectives are clues that this is all that is required. In contrast the remaining objectives are written using language including 'interpret', 'construct', and 'apply'; these suggest a need for functioning knowledge. For example, to model software using the UML requires an understanding of its abstract notational elements and their mappings to concrete entities. Similarly, applying patterns requires students to consider alternatives and to use their judgment. In both cases, students need to use higher-level cognitive processes. It is thus objectives 2, 4, 5 and 6 where careful consideration of teaching and learning activities is necessary to ensure that students exhibit the deep understanding sought.

3.2 Key design decisions

The course objectives convey *what* we intend students to be capable of following the course. We now turn our attention to *how* the course can deliver these outcomes. Essentially, this means making basic decisions concerning teaching and learning activities and assessment tasks. We made four high-level decisions:

- *Allow students sufficient time to build their knowledge structures.* Acquiring functioning knowledge requires that students develop their semantic structures, making connections between individual chunks of knowledge, iteratively. This process inevitably takes time.

The course's predecessor ran in a 5 week 3 lectures per week format with loosely coupled lab sessions shared with other courses. One observation of this condensed format was that students had little time to assimilate course material and then do any non-trivial design and implementation. In light of our first decision, to make sufficient time available to students to acquire the necessary depth of understanding, we restructured the stream. The new course now runs over 10 weeks with 4 contact hours per week per student. Contact hours are split 2 hours for a classroom session and 2 hours for a tightly integrated and supervised lab session where students implement designs formulated in the classroom, work on assessment tasks and reflect on concepts introduced in classroom sessions.

A basic feature of the new course is that each student has a formal learning partner who they work with on all coursework assessment tasks. The intention is for students not to divide up the tasks between them but to engage together and reap the benefits of peer learning. For example, they can talk over how best a design problem can be solved using a particular pattern, discuss modelling and design concepts, and generally resolve any misconceptions. We elaborate further on the range of active and peer learning activities used in the next section. Similarly, in Section 5 we provide the details of formative assessment for both coursework *and* exam.

With Open Source projects now very popular, there is no shortage of candidates to consider for inclusion as course subject matter. In addition to the advantages already given for using existing software, many Open Source projects are designed using patterns. This is appealing for this course since the product can also serve as an exemplar for patterns. Having looked at a number of projects, we opted for JUnit (Gamma & Beck 2001) which is a unit testing framework that has proven very popular in both industry and academia. JUnit is sufficiently large for our purposes, involving half a dozen packages of which two need to be understood to do the main coursework tasks, without being overwhelming for students. Furthermore, it is renowned as a fine exemplar of pattern synthesis. Finally, JUnit involves minimal overhead to introduce in the course since students are familiar with black box testing and JUnit is very easy to use.

4 Teaching and learning sessions

Using solely traditional lecturing for the classroom sessions would be ineffective, particularly because students would be impervious to the majority of information presented over 2 hours. For this reason alone, a mix of teaching activities is more suitable. Furthermore, since we have made a decision to embrace active learning, in the interests of fostering deep functioning knowledge, the classroom sessions also involve active learning activities.

The classroom sessions are organised by 3 logical units: fundamentals, design patterns, and advanced programming topics. The fundamentals unit includes a review of object orientation, an introduction to modelling using UML, and basic design principles. The unit on design patterns covers 4 design patterns in depth and the last unit is concerned with multithreading and socket-based inter-process communication. In all cases, the material is necessary to meet course objectives and for students to complete the assessment tasks.

4.1 Active and peer learning

What-if questioning often takes little time to do and results in feedback to teachers and students alike. For example, we use this when introducing UML class diagrams and in particular relationships and multiplicity constraints. Using constraints, modellers can be expressive in terms of the number of instances of participating classes that are allowed to be related. By presenting a class diagram and asking students what the multiplicity constraints really mean engages students; they have to interpret the diagrams and assess their own understanding rather than sitting passively where it is too easy for students to think they have understood. When we see that students have understood, we can change the constraints; subtle changes on a diagram can have significant changes in meaning. Again we use further questions to assess class understanding.

For example, given a simple class diagram involving two classes, `Lecturer` and `Course` and a `Teaches` association between them, we might ask whether a lecturer would be happy where the association is annotated with a `0..*` constraint at the `Course` end of the association. Would any lecturer reading this paper like a contractual clause stating that there is no upper bound on the number of courses they can be asked to teach?

Role play also engages students in their learning. In reviewing the basic ideas of object-orientation we have students acting as objects and playing out scenarios showing how links between objects are formed dynamically and how messages are processed. This approach has much in common with the Class Responsibility Collaboration (CRC) method (Beck & Cunningham 1989). Role play leads naturally into documenting scenarios using UML; once students have exercised a scenario, they document it using a UML object interaction diagram. From an initial class diagram and an interaction diagram generated from role play, we are able to explore the issue of well-formed models, where different views should ultimately be mutually consistent.

From earlier experience in teaching UML, taking the form of more traditional lectures to present abstract models, many students had difficulty in distinguishing between the different relationships on class diagrams. To address this, following a short presentation on relationships, a learning activity follows where

students reverse engineer a class diagram from a small set of source files. Following the exercise, we can assess students' understanding and probe at why a particular relationship is a dependency and not an association for example. Functioning knowledge and fluency in UML is not only necessary to satisfy course objective 2, but in addition is a prerequisite for understanding patterns since UML is the language used to communicate them.

To prepare students for what is in effect pair programming in extreme programming (XP) (Beck 2001), sharing a machine in the lab sessions and doing the coursework tasks with their learning partner, it is important that each student sees that they have a well defined role. Similarly to XP, one individual has ownership of the computer and types code while the other is concerned, in part, with source code inspection. The idea is that a rich dialog emerges between the two individuals; this is clearly a manifestation of peer learning where students benefit from their collaboration. In a classroom session, we engage students in carrying out a code inspection of a small number of source files using a prepared checklist of inspection items, based on work by Kolawa (2000). Rather than telling the students how effective inspection can be, they actually experience this for themselves and identify defects. We encourage students to reflect on the time it would take to find the defects using testing, as opposed to a relatively quick inspection.

4.2 Problem-based learning

Previous experience with teaching design patterns has revealed that simply picking out a particular pattern from the gang of four's book (Gamma, Helm, Johnson & Vlissides 1995), or similar text, and presenting it in the well know format (intent, motivation, applicability, structure, . . . , known uses) is ineffective for teaching undergraduates. At this stage in their education, students lack design experience and typically cannot see the value of a pattern. Books like the gang of four's are great references for experienced designers. This is because they have first-hand experience of design problems and can compare a pattern's solution with their own that may have consumed a lot of effort in its formation.

For students, however, who are being exposed to patterns for the first time, we believe they should engage in real problems where the pattern can help. In addition, we think that students should experience the full lifecycle of a pattern, from problem analysis through to implementation. This is necessary for students to really appreciate the benefits (and costs) of using a pattern and to validate their application of the pattern. To acquire the performative level of understanding sought, these are essential learning activities. To some extent though these activities have conflicting requirements; real problems tend to be too large and complex to implement within the time available.

In teaching the subset of patterns on the course, we do so using our own teaching pattern:

1. *Immerse students in a real problem.* We begin by introducing students to a real design problem. As a group, we explore possible solutions and analyse them with respect to design objectives. At this stage, we are not concerned with patterns and the intention is to identify solutions that can be subsequently improved using patterns.
2. *Present a solution to the problem.* We then present a solution that solves the problem, un-

knowingly to students, using a design pattern. This solution is compared to earlier solutions and is almost always a superior solution with respect to the design objectives of flexibility and reuse.

3. *Present the pattern.* We conclude the first hour of the session by introducing the pattern, largely in the established catalogue format mentioned earlier. Prior to the mid session break, we also present a new problem, on a smaller scale, which students should solve by applying the pattern.
4. *Engage students in applying the pattern to a small problem.* Students return and work in small groups, be it in pairs or slightly larger groups, on solving the problem set immediately before the break. Students are directed in their approach by a set of questions.
5. *Reflect on the pattern.* In the remaining time, students are invited to outline their solutions. It is invariably the case that different solutions are possible when applying patterns, and this provides a natural point of discussion on which variation of a pattern is most appropriate in the problem context.

Examples of real problems that we have used in step 1 include how to design a database access framework that is vendor independent and how Sun's engineers have designed the Swing framework to work, unmodified, with existing application classes. We invite students to suggest solutions and the class as a whole to critique them. In many cases, we offer solutions ourselves, consisting of UML models and code fragments, and allow them to be critiqued so students do not feel their own suggestions are being selected for criticism. There are often good aspects of their designs to point out in any case.

With the database access problem, we illustrate Sun's JDBC solution which draws heavily on the Factory Method pattern. Again, a combination of UML models and code is used to present the pattern-based solution. With the JDBC framework, UML models are effective at highlighting the interfaces that database vendors need to implement and the interfaces that client applications commit to. Source code fragments for a client application demonstrate that a client need not have any knowledge of the database vendor it uses. With the Swing framework, we show how the Adapter pattern has been used.

In contrast to the motivating problem, the problem tackled in step 4 is deliberately kept small to be implemented in a lab session. Implementing the pattern contributes to a student's understanding of it, helps students validate their design, and reinforces traceability from models to code. Taken overall, the course exposes students to a substantial problem that has been solved by the pattern, a small problem that students fully implement, and finally students meet the pattern a third time when applying it to enhance JUnit as part of the main assessed tasks.

The small problem for the Adapter pattern involves students reusing collection classes from Java's `util` package to implement a prescribed `OrderedCollection` interface with two implementations, a FILO and a LIFO implementation. Students have to make decisions as to which variant of the Adapter pattern to apply (object adapter or class adapter) and which `java.util` class or interface to play the adaptee role. Many students choose to use the class adapter and inherit from `java.util.Vector`, the adaptee; others use the object adapter with `Vector` again as the adaptee;

while others advocate the object adapter with an interface such as `java.util.List` as the adaptee.

We often ask students to write their designs using UML diagrams on OHP slides, some of which are selected for airing towards the end of the session during discussion of alternatives. Each design option provides for a rich discussion and allows us to tease out with the students that the third option is in general the preferred design. Briefly, the first option can result in subverting the integrity of an `OrderedCollection` implementation, since methods inherited from `Vector` can be called to add and remove items from any point. The second design does not suffer from this problem since the adaptee is encapsulated inside an `OrderedCollection` object, but the adaptee class is fixed at compile time. The third option is the most flexible since an `OrderedCollection` can adapt any class that implements the `List` interface.

5 Assessment

With the original design course, assessment was split 70% exam and 30% coursework. To reflect the practical emphasis of the new course, we changed this to a 50/50 split whilst still satisfying University regulations.

Task 1 Apply a combination of the Factory Method and Singleton patterns to develop a new version of JUnit that allows any `TestListener` product to be registered at runtime with the JUnit processing engine.

Task 2 Reuse an existing class, that handles persistence of test case results and which provides statistical functionality, by applying the Adapter pattern. The resulting class implements the `TestListener` interface.

Task 3 Use the Proxy pattern to implement a protection proxy for the result of task 2. This ensures that task 2's adapter is thread-safe in preparation for deployment in a concurrent server.

Task 4 Design and implement a remote proxy that sends requests to an instance of the class resulting from task 3 which executes in a server process. The proxy encapsulates the details of distributed communication.

Table 2: Main coursework tasks

5.1 Holistic assessment for coursework

The coursework tasks are split in two: *supporting* tasks and *main* tasks. The former require students to apply patterns in a simple and self-contained context to build their understanding of individual patterns. The latter involves subsequently extending JUnit with new functionality using patterns that students have gained familiarity with in the supporting tasks. A representative example of a supporting task is applying the Adapter pattern to reuse `java.util` classes when implementing the `OrderedCollection` interface, as discussed earlier.

The relationship between supporting and main tasks is important as it is an enabler for students to

develop and evolve their knowledge networks. Students start off with the simple supporting tasks, gain confidence in using a pattern and are then motivated to tackle the more complex tasks. This is consistent with education theory of Feather (1982), who points out that motivation grows with confidence, and Race (1993) whose model recognises that motivation is at the core of any learning activity.

There are 4 main tasks which involve developing JUnit into a distributed client/server application. Essentially, enhanced JUnit clients send the outcomes of running test cases to a central server. The data on the server can then be analysed to identify troublesome parts of an application, indicated by test cases that regularly fail. Table 2 gives a brief overview of the tasks. The `TestListener` interface referred to in the table represents an intended extensibility point of the JUnit framework. `TestListener` can be implemented by any class whose instances have an interest in the outcome of running test cases.

All coursework assessment tasks are assessed *holistically*. Holistic assessment (Biggs 1999) recognises that the whole is greater than the sum of its parts. Biggs uses an anecdote drawn from medical school to motivate the need for approaches like holistic assessment.

For a medical task, assessment criteria might include correct diagnosis of a patient's condition, in this case necessitating an operation. Further criteria are for making an incision correctly, performing the actual operation, and closing the wound afterwards. Marks are awarded for each criterion depending on how well a student satisfies it. A student is deemed to exhibit sufficient functioning knowledge if the sum of marks awarded exceeds some specified threshold.

This example brings into question the use of simple mark summation that is commonplace in universities. The problem in this case is that a student will pass the task if they carry out all stages perfectly but leave an instrument inside the patient! With holistic assessment, *all* stages must be performed satisfactorily for a student to pass.

We have adopted a variation on holistic assessment. To illustrate this, consider the criteria for main task 2:

- *A UML class diagram and an object interaction diagram.* These should be well-formed and show appropriate application of the Adapter pattern.
- *Conformance to design.* The implementation should be consistent with the design.
- *Correctness of implementation.* The implementation should meet its specification and correctness should be demonstrated by JUnit test cases.
- *Verbal understanding.* Both partners should be able to demonstrate they have understood what they have done.
- *Use of packages and organisation of software artefacts.* A package should be used and source and executable files should be stored separately.

Each criterion carries one mark and students are awarded the mark only if they satisfy the criterion. If a student satisfies all criteria, their mark is doubled. Thus in the case of task 2, the maximum mark a student can be awarded is 10. If they satisfy 4 of the 5 criteria, they get a mark of 4 out of 10.

Thus, students are encouraged to meet all criteria and are rewarded where they satisfy the wholeness of the task. Conversely, they are penalised where they do not. Holistic assessment is appropriate for our course because of the associated support framework for students. Specifically, work is assessed formatively and can be repeatedly submitted for assessment in any lab session. Students receive verbal feedback for each criterion, and are made aware of which criteria, if any, they have failed to satisfy. Students are thus able to address any problems and subsequently re-demonstrate their work.

Other factors which contribute to holistic assessment being appropriate are that all assessment criteria are explicit and that software itself conforms to the holistic principle. Assessing students holistically works only where assessment criteria are transparent to students since they know exactly what is expected of them and it takes little time to explain to a student which criteria they have yet to meet. For the latter factor, software that is well designed, documented and tested is significantly better than software that meets its functional specification but which is not maintainable.

Returning to the assessment properties introduced in Section 2, the coursework tasks are valid, reliable, and rigorous. The tasks are *valid* since there is traceability from the learning outcomes, presented in Section 3 and explicit assessment criteria of individual tasks. For example, the first criterion of task 2 is derived from learning outcomes 2 and 4. Since criteria are explicit, the tasks are *reliable*; this is particularly important for this course because several people are involved in assessing students and offering feedback in the labs. It is obviously important that students are treated consistently and fairly.

As described earlier, all tasks are performed in pairs, yet the tasks remain *rigorous* because of the verbal understanding criterion. To satisfy this criterion, students must show they have understood what they are demonstrating. Doing so requires each student within a pair to answer a question posed by an assessor. To retain reliability, all assessors share a common bank of questions to draw on. Further, this criterion can be used to discriminate between individuals within a pair where it is obvious that one has the deep level of understanding sought while the other is free riding.

A final point to note on our coursework assessment process is that the cost on assessors giving feedback is low and contained within scheduled sessions. All coursework tasks are marked in the lab sessions which means that there is no additional marking of coursework outside of the labs. Moreover, feedback to students is of much value and is instantaneous.

5.2 Formative examination

Similarly to our approach to coursework assessment, the way we assess students using the exam component is also unconventional. We have opted for an alternative to the traditional exam for two reasons. First, software design is an iterative process that requires creativity and reflection. As we constantly remind students, design also involves considering options and making sensible tradeoffs. These characteristics of the design process mean that it is very difficult to assess under the artificial conditions, particularly timing constraints, of the traditional exam paper. The second reason for looking for an alternative is that the traditional exam is a summative form

of assessment and consequently its value to students is very limited.

One might argue that a traditional exam would actually be appropriate as the coursework assessment tasks measure students' functioning knowledge. Using this argument, we could perhaps justify the use of exam questions that seek mainly declarative knowledge, for example asking students to describe the intent of a particular pattern, its benefits, and its drawbacks. Of course it would also be possible to write more problem-oriented questions, but the exam's timing constraint is a limiting factor. Another argument would be to assess students 100% by coursework. For many universities, however, regulations may preclude this particularly where the coursework is carried out in pairs or groups.

We have retained the exam component and used it as the ultimate test of students' functioning knowledge in software design. While the coursework tasks assess students' functioning knowledge of a handful of patterns introduced in classroom sessions, the exam task assesses students' ability to solve a new problem using a pattern which has *not* been covered on the course. Essentially, students are presented with a problem prior to the exam and are expected to prepare for the exam by finding and applying a pattern to solve the problem. Similarly to traditional exams, the question remains unseen, closed book and time constrained. However, students know the problem on which they will be assessed.

This year's exam problem was concerned with part of a security package, in particular access control. Students were presented with a design for representing different types of user with a superclass named `Operator` and subclasses to represent specific classes of user, such as `Administrator`, `Student` and `Lecturer`. Each subclass overrides the `isAuthorizedTo` method which it inherits from `Operator`. This method takes an argument describing a particular action, and the method returns true or false depending on the action and implementation class.

Students were made aware that in reality, different individuals can play multiple roles. To accommodate this, additional subclasses are introduced, one for each combination of roles. Clearly, this leads to an unmanageable class hierarchy as the number of classes required grows exponentially with roles. Furthermore, code is duplicated in classes which leads to maintainability problems. For example, to change access rights for lecturers, classes `Lecturer`, `LecturerAndStudent`, and `LecturerAndAdministrator` must all be changed.

In the problem description given to students, the document is written to include vocabulary that when used as search terms in Google results in a host of articles on the Decorator pattern. For example, "subclass explosion" plus "design pattern" returned 29 hits, the majority of which were useful. In introducing patterns earlier in the course, a JavaWorld article (Geary 2001) is identified as recommended reading; this article includes the Decorator pattern and illustrates it using the flexibility of Java's input/output stream classes. Students are encouraged to work with their peers in developing their preparatory work and to implement their designs to validate them. Since the problem is similar in scale to a supporting task, implementation incurs little time. Students are also invited to get feedback from staff on their designs at scheduled times prior to the exam. This coupled with collaboration with peers on preparatory work makes the task a formative form of assessment.

The exam component satisfies all three require-

ments of assessment tasks. The exam task is *valid* since it is aligned with course objectives. The problem seeks functioning knowledge with respect to modelling and design, therefore addressing outcomes 2 to 4. The task also requires students to think about design objectives (outcome 1), in particular remedying the lack of extensibility and scope for error with the original design. This task does not cater for outcomes 5 and 6 but these are assessed in the coursework tasks. Simply providing a marking scheme ensures that the task is *reliable*, but this should allow for diversity in students' solutions. Finally, the exam task is *rigorous* because students have no knowledge of the question prior to the exam. Rote learning a peer's solution would be of only limited value because students must understand their solution in order to answer the exam question.

6 Evaluation

This year, there were 68 second year students, all of whom studied the Software Design course. The coursework tasks are a mandatory component for all students, but due to characteristics of the Software Engineering stream, students can choose to opt out of the exam component for this course. Briefly, there are three courses within the stream, one of which is an introduction to Software Engineering and the other is concerned with interactive systems and focuses on a mix of HCI and technical issues for developing GUIs. Students select 2 of the 3 stream components on which to be assessed by examination; 52% of students opted for the Software Design exam question.

6.1 Assessment results

Figure 1 shows the distribution of results for the supporting tasks, main tasks and exam. The distribution curves clearly all have a different profile to the bell-shaped curve, where most students score around 50–60%, which is often sought by traditionalists. We question why this should be the standard model to follow since aligned assessment tasks should measure the extent to which students satisfy learning outcomes. Ideally, we would like the majority of students coming out of a course and demonstrating that they have satisfied all outcomes. As Figure 1 shows, the distribution of marks for the supporting tasks is much closer to our preferred distribution, with 54 of the 68 students scoring 100%. Of the remaining students, 6 exceed 70%, 4 are in the 50% band, leaving 4 fails.

The very high scores for supporting tasks are perhaps not surprising since the tasks are generally small problems and partially discussed in classroom sessions. For the main tasks, nearly 50% of students demonstrated that they satisfy all criteria, resulting in a score of 100%. Of the remaining students, 28 scored highly in the 60–90% bracket, leaving a minority of 13 students of whom 8 failed to achieve a pass mark. Similarly to those who performed poorly on the supporting tasks, these latter students had poor attendance records.

Exam performance involves the least variance, with all students who sat the exam question bar one achieving a pass mark. The vast majority of students scored within 60–90% with 4 students achieving a particularly impressive mark of 100%. From our observations, the increase in consistency of the exam marks over the main task marks is due to the subset of students who opted for the exam question being particu-

larly interested in the course and motivated to do the preparatory work.

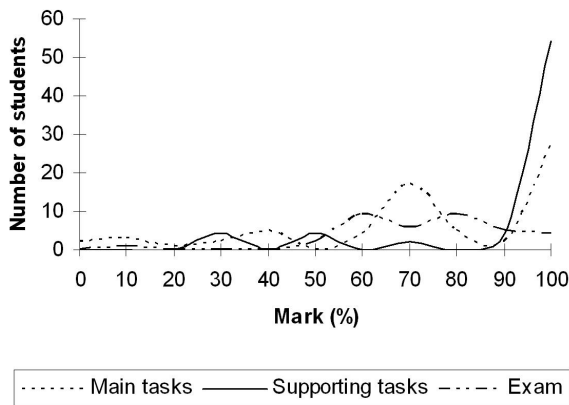


Figure 1: Student assessment results

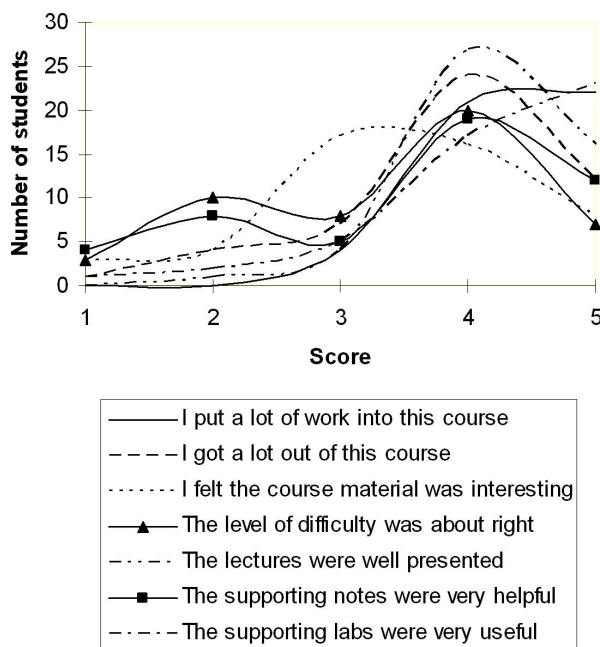


Figure 2: Student questionnaire results

6.2 Feedback

Of the 68 students who took the course, 48 completed and returned the standard University end-of-course questionnaire. The questionnaire includes 7 statements each of which students score in the range 1–5 where 1 means that they strongly disagree, 3 indicates they have no firm opinion and a 5 represents strong agreement. Figure 2 shows the distribution of responses for each of the 7 statements.

The majority of students agreed, many strongly, with the statements “I put a lot of work into this course” (43), “The supporting labs were very useful” (40), “The lectures were well presented” (43), and “I got a lot out of this course” (36). The remaining three statements are more contentious. Of the 48 students who returned the questionnaire, 17 had no firm opinion on whether or not the course was interesting, with 24 believing it was and 7 thinking it was not. The

statement “The level of difficulty was about right” divided students, with responses spanning 1 through 5. However, a significant minority of 20 agreed with this statement. Similarly, students’ response to the statement “The supporting notes were very helpful” was mixed.

In addition to the quantitative feedback, students were also invited to identify two things that they liked about the course, two ways in which they feel the course can be improved, and to offer any other comments. Abstracting from students’ responses, four generally positive themes were identified. Most notably, students remarked on the availability of staff and feedback in practical sessions. This is consistent with the quantitative feedback we received where half of the students responded to the “The supporting labs were very useful” statement with a score of 5 and most others with 4. This can partly be attributed to helpful and knowledgeable lab staff, but also because of the relatively high staff/student ratio that results from having students paired. The ratio effectively doubles. In the case of this course, students were split into two lab groups with 34 students in each group. This gives 17 pairs per lab and with 3 members of staff running a lab, a ratio of 1 member of staff for every 5 student pairs.

Following closely, the second theme highlighted was formative assessment. Several students commented that they liked the incremental style of the coursework and the relationship between supporting and main tasks. This confirms to us that students do learn as theorists such as Feather (1982) and Race (1993) suggest and that starting with simple problems builds confidence and motivation for subsequently tackling more complex problems. With respect to the exam, a number of students also said that they preferred the approach adopted on this course. In particular students’ reasoning included that this style of exam “better mirrors reality”, “allows for more preparation time” and “reduces anxiety”. Contrary to these comments, however, one student favoured a more traditional approach stating that valuable revision time is taken away from other courses. We disagree since the preparation focuses on a subset of course material; students do not have to be concerned with remembering details about unrelated patterns for example.

Third, a number of students remarked favourably on the practical emphasis of the course. Students liked the depth of material covered, the “real world” design examples, “advanced design techniques” and working on a “real” product. From talking to students in the lab sessions, it also became clear that some were finding patterns useful in other courses, particularly two group-based courses that attract among the best students in the year and involve significant software development. The final theme was peer learning. Most students saw this as a beneficial experience, although a small minority had reservations principally because of free-riding partners and general problems with meeting up with their partner. This phenomena, involving only a small number of students, seems to be a recurring and unfortunate reality in university education.

Students’ qualitative feedback was also consistent with the quantitative responses to the “The level of difficulty was about right” and “The supporting notes were very helpful” statements. With the former, a number of students commented that they considered the workload to be too high, perhaps borne out by a significant minority of students (18) not completing the final main task. This is important for us to

note since overloading students can have a negative impact on student motivation (Biggs 1999). Whether students were overloaded is a subjective matter, however, and half of those who completed a questionnaire and who expressed an opinion considered the workload to be about right. Regarding the supporting notes, a small group of students also stated that they found them confusing and hard to follow. While a conscious effort was made to make the notes informative, it is difficult to escape the use of jargon to retain precision. In addition, coursework descriptions were very detailed, containing useful hints, and perhaps some students saw the length of these documents as off putting.

Interestingly, the topic of holistic assessment attracted neither negative nor positive feedback. This form of assessment simply seems to have been accepted as a natural characteristic of the course.

7 Discussion and conclusions

In this paper, we have described our experience of designing and running an undergraduate course in software design. The approach used to design the course has in common with a core part of its subject matter, design patterns, the application of proven techniques. For design patterns, these proven techniques lead to software that exhibits desirable qualities like reuse and extensibility. With our course, fundamental tried-and-tested techniques have been employed to equip students with deep functioning knowledge that enables them to do design, modelling, and implementation effectively.

We have reviewed relevant education theory and have identified a number of techniques for making teaching, learning, and assessment more effective. Much of this material should come across as common sense and will be familiar to many readers, who, like us have engaged in the educational literature. For many other readers, we have simply labeled things they already do; this is reassuring. For yet others, we hope Section 2 provides a concise introduction to good teaching practice.

We have also demonstrated how generic education theory and principles can be applied to teach Software Engineering. In short we have applied constructive alignment, active learning, peer learning, and formative assessment. Examples of our application of active learning range from short changes in activity when lecturing, to sessions that are driven by interactive and problem-focused activities; the latter principally as a means for students to acquire functioning knowledge of design patterns. Peer-learning is manifested by students having a formal learning partner who they pair-program with, and with who they collaborate on coursework tasks. In addition, students work in small groups on problems in the classroom.

Our application of formative assessment is particularly novel. We have introduced holistic assessment that encourages students to meet all criteria of coursework tasks; to do well students cannot escape any one element of a coursework task such as design documentation or testing. Feedback on coursework tasks is useful and delivered in a timely fashion to students. Furthermore, assessing coursework imposes a low overhead on staff running the course and does not consume time outside of scheduled labs. These ideals often remain merely so where other assessment practices are used. For the exam component, we have devised an approach that is highly appropriate for measuring students' functioning knowledge and which

is also formative yet retains rigor.

The active learning paradigm does have a cost however. In this author's experience, developing problem-oriented sessions requires care and involves more time than preparing conventional lectures. Problems need to be well thought through, and in the case of software design, with a mix of candidate design options that exhibit a range of properties to facilitate rich discussion. Preparing such problems and design options requires creativity and iteration, which incidentally are the characteristics of software design that led us away from using the traditional exam for assessing students.

In addition to an increase in preparation time, active learning trades breadth in favour of depth, simply because engaging students in doing activities takes time that could otherwise be spent covering additional material. However, in the context of our design course, depth is preferable since a broad but shallow treatment of the subject matter will not result in students with deep functioning knowledge. Furthermore, depth of knowledge is the basis of transferable skills. The majority of students who sat the exam question performed very well and demonstrated their ability to apply material not covered in the course to solve a new problem.

Using JUnit as the subject of the main coursework tasks proved popular with many students since it is a real piece of software. JUnit is also sufficiently small yet sufficiently interesting, being a rich exemplar of design pattern synthesis. These characteristics are important for a short introductory course; the former so as not to overwhelm students and the latter so as to provide a realistic example of the effective application of patterns.

Student results and feedback on the whole have been encouraging. Given that the assessment tasks are aligned and rigorous, we are confident that the results are an accurate indication of students' abilities in modelling, designing, and implementing relatively large software projects. Our one concern from student feedback is the possibility that the workload is too high for some students. However, all students who participated did very well on the supporting tasks and did similarly so on at least 3 of the 4 main tasks. The breakdown of coursework into supporting and main tasks, coupled with the incremental nature of the latter establishes an effective safety net.

8 Acknowledgments

We would like to thank Lynne Blair, for her feedback concerning active learning, and Jamie Hillman for his assistance in running this course. We would also like to thank Rebecca Marsden and Ruth Stalker for their feedback as peer observers, and Nick Blundell and Craig Morall for their assistance in running lab sessions. Finally we would like to thank Cath Ewan for her dedication to course administrative activities.

References

- Beck, K. & Cunningham, W. (1989), A Laboratory for Object-Oriented Thinking, *in* 'International Conference on Object Oriented Programming, Systems, Languages and Applications', New Orleans, Louisiana, USA, pp. 1-6.
- Beck, K. (2001), *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

- Biggs, J. (1999), *Teaching for Quality Learning at University*, SRHE (Society for Research into Higher Education) & Open University press.
- Blair, L. (2000), Learning, Assessment and Student Diversity: An Eternal Golden Braid?, CiLTHE Stage 2 Document, Lancaster University.
- Bligh, D. A. (1971), *What's the Use of Lectures*, Penguin Books.
- Bloom, B. S. & Krathwohl, D. R. (1956), *Taxonomy of Educational Objectives*, Longmans.
- Boud, D. (1995), *Enhancing Learning Through Self Assessment*, Kogan Page.
- Braude, E. (2003), *Software Design: From Programming to Architecture*, Wiley.
- Feather, N. (1982), *Expectations and Actions*, Erlbaum.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable software*, Addison-Wesley.
- Gamma, E. & Beck, K. (2001), *JUnit: A Regression Testing Framework*, <http://www.junit.org>.
- Geary, D. (2001), 'Amaze your Developer Friends with Design Patterns', *JavaWorld*, October, <http://www.javaworld.com>.
- Honey, P. & Mumford, A. (1995), *Using Your Learning Styles*, Peter Honey Publications.
- Kolawa, A. (2000), 'Coding Standards in Java: Do We Need Them? Six common Java pitfalls and how to prevent them', *The Java Report*, March, <http://www.adtmag.com>.
- McKeachie, W., Pintrich, P., Lin, Y. & Smith, D. (1986), *Teaching and Learning in the College Classroom*, University of Michigan.
- Race, P. (1993), Never Mind the Teaching, Feel the Learning, in 'SEDA (Staff and Educational Development Association) paper', No. 20, <http://www.seda.ac.uk>.
- Warren, I. (2002), *Migrating to a Teaching Style that Facilitates Active Learning*, CiLTHE Stage 1 Dissertation, Lancaster University.