First Year Programming: Let All the Flowers Bloom

Raymond Lister and John Leaney
Faculty of Information Technology
University of Technology, Sydney
PO Box 123, Broadway 2007, New South Wales
{raymond,johnl}@it.uts.edu.au

Abstract

We describe our criterion-referenced grading scheme for a first year programming subject, which has been designed to allow all participating students to achieve their full potential. Traditional norm-referenced grading schemes, where all students work on the same assessment tasks, result in tasks that may be effective for the middle-achieving student, but the tasks do not allow the weakest students to engage effectively, nor do these tasks stretch the strongest students. Our criterion-referenced scheme uses a mix of several assessment strategies to overcome this problem. The strategies have been used before in traditional assessment environments, but in isolation, whereas we combine the strategies into a coherent, explicit grading philosophy based on Bloom's taxonomy.

Keywords: first year programming, criterion-referencing, Bloom's Taxonomy.

1 Introduction

The goal of every university teacher should be to realise the potential of each student. When teaching a first course on programming, however, we more commonly achieve the opposite: The stronger students are not challenged, while the weaker students flounder.

The literature on the teaching of programming contains abundant reports of encounters with the weak students. Results from a recent project by McCraken *et al.* (2001) are compelling, because of the number of authors from differing educational institutions and cultures. The 10 authors teach introductory programming across 8 universities, in 5 countries. Each author tested his/her own students on a common set of programming tasks. The students performed much more poorly than the authors had expected. The students did not simply fail to complete the set task, most students did not even get close to solving the task. Whereas many other similar reports, from authors working alone in a separate institutions, might be dismissed as the result of poor teaching skills, this multinational project suggests that there is something fundamentally wrong with the way we, as a discipline, teach programming.

The problem is not simply that students cannot program after completing their first course. The problem is three-faceted: (a) our goal is that students should be able to program, (b) our weaker students cannot, (c) but we pass them anyway. One solution would be to fail those weaker students who do not satisfy our goal, but the very high resultant failure rate would be unacceptable in most institutions. Perhaps then we should reconsider our goal for the weaker students, in their first programming course.

The move to teaching object-oriented programming has made it even more important that we re-evaluate our goals in teaching. Weak students now must cope with the concepts of object-oriented programming in addition to the problems they already faced with structured programming concepts taught using 3rd generation languages.

And while we focus our time and energy on the weaker students, what becomes of the stronger students? In the multinational project of McCraken *et al.* (2001), student performance was bimodal. That is, while most students performed poorly, a smaller percentage of students performed quite well. If we simply water-down introductory programming courses, to accommodate the weak students, the strong students will go unchallenged.

Neither weak nor strong students achieve their true potential in norm-referenced grading schemes. In pure norm-referencing schemes, students attempt the same task, an order of merit is established based on the performance of each student, then all grades (including the failing grade) are assigned according to some desired grade distribution. Such a scheme does not guarantee minimal competency in the weaker passing students, or guarantee that the best students have been challenged.

Weak and strong students are better accommodated by criterion-referenced grading schemes. In pure criteria-referencing, explicit clear criteria for each grade are communicated to students. A grade is assigned to each student according to the criteria satisfied by the student, irrespective of the resultant grade distribution.

1.1 Bloom's Taxonomy and the Three Grades

When devising our grading criteria, we applied Bloom's (1956) Taxonomy of Educational Objectives. The taxonomy contains six levels. Each successive level depends upon behaviours acquired at the earlier levels. When describing each of these levels, it is common practise to include a list of words illustrating the behaviours that a student is expected to exhibit. The levels, from lowest to highest are:

Knowledge: the recall of specifics and universals, the recall of methods, and processes, or the recall of a pattern, structure, or setting. Bloom's word list of activities includes "*memorise", "name", "recognise"* and "*relate".* For example, a suitable assessment question for Java programming at the knowledge level is *"State the three types of loops in Java" or "Which type of Java loop always executes at least once?"* (As will be explained below, in our criterion-referenced system, we do not actually assess students at the knowledge level. We begin assessing students at the next level, "comprehension.")

Comprehension: a type of understanding when the individual knows what is being communicated and can make use of the materials or idea being communicated without necessarily relating it to other material or seeing its fullest implications. Bloom's word list of activities includes "*restate"* and *"translate".* A suitable assessment task for Java programming at the comprehension level might involve the translation into a single Java statement of a very specific instruction from either English or pseudo-code. For example, "Write a Java statement that declares a variable 'x' that may contain integer values".

Application: is the use of abstractions in particular and concrete situations and may include general ideas, rules of procedures, generalised methods, technical principles, ideas, and theories that must be remembered and applied. Bloom's word list of activities includes *"calculate", "solve",* and *"write".* We believe that many traditional programming assignments are tasks at the application level, provided the assignment task is quite specific. Suppose a student was required to write a class, and the class had been specified down to a precise description of the method headers and data members. We would regard that as an application level task, provided the code within each method was not long (eg. <20 lines) or algorithmically complex.

Analysis: emphasises the breakdown of the material into its constituent parts and the detection of the relationships of the parts and of the way they are organised. Bloom's word list of activities includes "*categorise", "differentiate", "discriminate" and "distinguish".* For example, a suitable assessment task for Java programming at the analysis level might involve providing the student with code that implements the Model-View-Controller pattern, and asking the student to identify the parts of the code that implement the Model, the View, and the Controller.

Synthesis: the putting together of elements and parts so as to form a whole. Bloom's word list of activities includes *"create", "design", "organise",* and *"plan".* As with the application level of the taxonomy, we believe that many traditional programming assignments are tasks at the synthesis level. The difference from an application task is that a synthesis task contains more sophisticated design choices. Many programming textbooks contain the term "Problem Solving" in their title, indicating that students are not merely taught how to code in a particular language. In addition, students are taught (at least so the textbook authors allege) to take an initial vague problem description, refine it, decompose the program into classes, and determine appropriate methods in each class. Problem solving in this sense is a synthesis level task.

Evaluation: the making of judgments about the value, for some purpose of a solution, appraising the extent to which the solution is accurate, effective, economical, or satisfying. Bloom's word list of activities includes *"assess", "evaluate",* and *"judge".* This level of the taxonomy maps easily to the teaching of programming, as our discipline values clear, concise and efficient code. However, traditionally it is the teachers who evaluate the students' code. In the context of Bloom, "evaluation" is an activity the students perform, and teachers grade the quality of such evaluations. An explicit approach to "evaluation" would require the students to review some given code, and we would grade that review. Traditionally, our discipline assesses "evaluation" implicitly; under the assumption that students who design good programs are good at self-evaluation.

1.2 Six Levels into Three Grades

At our university, there are four passing grades, "Pass", "Credit", "Distinction", and "High Distinction" (or simply "P", "C", "D", "HD" respectively). In developing our grading criteria, we found it relatively easy to assign criteria to "P" and "HD". However, we struggled to devise criteria to distinguish between the middle two grades, "C" and "D". Eventually, we decided to assign the same criteria to the middle two grades. Thus, we effectively have three passing grades. We regard the traditional four passing grades as an historical accident, a relic of the British origins of our Australian university system. (The American system only has three passing grades.)

To develop the criteria for each of our three passing grades (after combining the C/D grades), we grouped the six levels of Bloom's taxonomy into three pairs, and applied each pair to those grades, and thus:

a) A "P" is awarded for performance at the knowledge and comprehension levels.

b) A "C" or "D" is awarded for performance at the application and analysis" levels. Which of the two grades is awarded depends upon the percentage mark achieved by a student.

c) A "HD" is awarded for performance at the synthesis and evaluation levels.

As discussed earlier, the traditional "problem solving" goal of an introductory programming course - that the students should be able to take a vague problem description and write a program - maps to the synthesis level of Bloom's taxonomy. Thus, our criteria for a "P" and "C"/"D" do not require students to meet that traditional goal; only "HD" students meet this goal. We will elaborate upon this process in the following sections of this paper.

As a faculty policy, we do not award a marginal passing grade, in some places known as a "terminating pass", or "conceded pass", for students who score just under 50%.

In this particular subject, we offer such students a supplementary exam. If they pass that exam they receive a "P" grade, otherwise they fail.

For most teachers, the distribution of grades indicates something about whether a subject was taught successfully, whether the subject was too hard or too easy. Table 1 shows the complete distribution of grades, for the first time we ran the subject in this criterion-referenced format, before and after the supplementary testing process. The first time this course ran with our criterion-referenced grading scheme, 29% of the class initially failed, but eventually only 10% were awarded an "F" after supplementary testing. Every university has its own conception of what is an acceptable failure rate (though not often is that conception committed to paper). The 10% failure rate in this criterion-referenced subject is higher than the failure rate in the three other subjects that our IT students do in their very first semester.

Grade	Before Supplementary	After Supplementary
F	29%	10%
P	18%	37%
C/D	47%	as before
HD	6%	as before

Table 1: The grade distribution before and after the supplementary tuition and testing.

1.3 Environment and Design Constraints

Before proceeding to explain our grading scheme in detail, we place our approach in its institutional context. Our first year class contains around 300 students, all of whom intend to eventually work in an IT career. Our university attracts both international and local students, drawing the latter from the top 13% of our state's high school graduates. Over half the class has written a program in some language before our subject, but few have written in Java, which is the language we teach.

The subject objectives are as follows (and while some readers may find these objectives problematic, they are given to us, and we are required to design a subject around these objectives):

At the end of this subject, a student will:

(a) Understand basic OO programming concepts of classes, instances, events, and methods.

(b) Understand basic program control constructs of sequence, selection and iteration.

(c) Read a program design and translate it into a readable, working program.

(d) Apply a systematic approach to testing and debugging;

(e) Express the basic principles of GUI design, from both the user and programmer perspective.

(f) Be able to express, in readable working code, the basic sorting and searching algorithms;

While we are obliged to remain faithful to the above objectives, we have edited them to make our learning goals for each grade more explicit. Traditionally, subject objectives are applied unaltered to all students in a class. The implication is that students achieving higher grades in some sense satisfy the objectives better than the students who achieve lower grades, but this is not made explicit. In our criterion-referenced approach, the above objectives are edited to form a set of objectives for each of the grades. We will provide the edited versions of the objectives in subsequent sections of this paper.

We have just completed running this subject in this new criterion-referenced format for the second time.

2 Knowledge/Comprehension and Grade "P"

Many teachers assume that students can only learn programming by writing programs. Clearly, students must write some programs, but do they learn best by almost exclusively writing programs, from the very beginning? We believe that, while stronger students do benefit from writing code from a very early stage, weaker students should be eased into the writing of programs.

Buck and Stucki (2001) observed that students who are required to write complete programs as lab assignments are "*overwhelmed, uncertain of how to begin, and grasping at the air* ... [leading] ... *to the self-destructive tendency to do experimental programming, where they just randomly throw things in to see if it helps.*" We teachers can (and often do) blame the students, and exhort them to be more principled (like the stronger students), but if we accept that this behaviour is normal in weak students we need to re-evaluate our assumption about writing code, in the context of weak students.

Four broad strategies for teaching programming have been identified by Fincher (1999). Not all of these approaches emphasise writing. One strategy, which Fincher calls "literacy", emphasises the reading of programs. (The chosen name for this strategy is unfortunate; many computing subjects intended for students who will not go on to an IT career are called "Computer Literacy". Our class consists entirely of students who intend to pursue IT careers.) The justification for this strategy is an analogy: when learning to write as children, and even when learning to write as adults in undergraduate literature classes, our teachers emphasised the reading of good examples. Fincher, as well as Kolling and Rosenberg (2001), and also Deimel and Naveda (1990), have suggested that the teaching of programming might place greater emphasis on reading. Even such luminaries of computer science research as Kernighan and Plauger (1981) subscribe to the maxim "*Careful study and imitation of good programs leads to better writing.*"

The lowest two levels of Bloom's taxonomy are "Knowledge" and "Comprehension", and they are broadly consistent with the "literacy" approach to teaching programming. Having made the connection between Fincher's "literacy" approach and Bloom's taxonomy, we edited the original subject objectives (given earlier) to form the following objectives for the "P" student:

If you successfully complete this subject to a pass level (mark of 50-64), you will have demonstrated that you:

(a) Can comprehend basic OO programming concepts of classes, instances, events, and methods.

(b) Can comprehend basic program control constructs of sequence, selection and iteration.

(c) Can read pseudo-code and translate it into a readable, working program.

(d) Know the basics of testing and debugging.

(e) Know the basic principles of GUI design, from both the user and programmer perspective.

(f) Can comprehend code that implements the basic sorting and searching algorithms;

To anyone not familiar with Bloom's taxonomy, the wording of the first two objectives is almost indistinguishable from the wording of the original objectives, but "comprehend" in the context of Bloom has a more precise meaning than the original use of "understand". The third edited objective is more specific than its counterpart in the original objectives. While "apply" is used in the fourth of the original objectives, it was written without Bloom's taxonomy in mind, and is therefore vague, whereas the use of "know" in the edited version is meant to invoke Bloom's taxonomy. Likewise, the fifth and sixth edited objectives are more specific than the original objectives.

2.1 Lab Exercises

Our lab exercises are short and simple, designed primarily as learning experiences, and are probably similar to many formative laboratory tasks in other programming subjects. When we first ran the subject with this criterion-referenced grading scheme, the lab exercises were intended for formative assessment only. That is, the lab exercises did not contribute to a student's final grade in any way. The lab exercises were provided as a learning experience. We are sympathetic to Roumani's (2002) view that lab exercises should be formative, as they can be a more effective learning environment than lectures.

In the second running of the course, however, we made some of the lab exercises summative. That is, to qualify for “P”, a student had to present working solutions to a subset of the lab exercises. (The exercises do not attract any marks. They are simply a pass/fail requirement.) We made this change for two reasons. First, some of our faculty colleagues were not happy with the original assessment scheme, arguing that a student could pass the subject without writing any code. From our contact with students, we feel this is unlikely, but it seemed politically prudent to acquiesce to our colleagues. Second, we actually feared the opposite of our colleagues. That is, we feared students were failing because they had not given themselves the benefit of learning from the lab exercises. The 29% initial failure rate, and its drop to 10% after supplementary testing, suggested that students were not engaging with the course until very late, when confronted with initial failure.

While some of the lab exercises are now nominally summative, our approach to marking is relaxed. We try to maintain the ethos of formative assessment. We merely want to see that a student has made a solid attempt, and learnt something in the process. Students may also make multiple attempts, and seek assistance from teaching staff. Also, students may work in groups of up to three, and learn from each other (although interestingly few students have elected to pursue the group-work option).

Students may work on the exercises outside the lab, and merely present the exercises at a lab session for marking. While the sincere student benefits from this *laissez faire* arrangement, there is considerable scope for cheating. That was a consideration in also having a lab exam.

2.2 Lab Exam

Anyone who has taught a programming-related course that follows an introductory subject has probably had the disconcerting experience of encountering a student who seems devoid of all fundamental programming skills, despite passing the prerequisite course. Plagiarism is a fact of life. The “P” grade must be protected, to ensure some minimal standard among students.

Approximately two thirds of the way through semester, we hold a one-hour lab exam. Since a "P" student is not required to write original code in this subject, the lab exam merely requires students to convert pseudo-code for a complete class into working code. Here are some examples of typical pseudo-code statements that students are required to translate into working Java:

- Import everything from the "javabook" package.
- Header for a class, called "SmallestNumber".
- Header for a public static method, called "main", which returns nothing, and which takes the parameter signature (String args[]).
- Declare an integer constant "SENTINEL" and set it to be zero.
- Declare a boolean variable called "firstTime" and set it to be true.
- Declare "mainWindow", an instance of "MainWindow", and call its constructor that takes the constant String parameter "Program SmallestNumber".
- Send a message to the "show" method within "mainWindow".

For the “P” student, mastering Java syntax is non-trivial, and we insist the code must work completely. Approximately 10% of the class fail this exam at their first attempt, but almost everybody passes after two attempts. The first time we ran the subject in this form, we promised a third attempt at the lab exam to anyone who passed the multiple-choice exam. However, and perhaps significantly, all students who failed their second attempt at the lab exam also failed the multiple-choice exam.

2.3 Multiple Choice Exam

Most teachers dismiss multiple-choice exams as being too easy, but with careful design it is possible to set demanding multiple-choice exams. The pass mark for the exam is set to 70%, a typical pass threshold for "mastery" exams, as we believe that the students must manifest a strong grasp of the material in this exam if they are to cope with the material in their next subject. Recall, from Table 1, that 29% of the class failed this exam on their first attempt … our multiple-choice exams are not easy.

The style of our multiple-choice questions has been described elsewhere (Lister, 2000 and 2001). These questions are designed to require knowledge and skills that only just fall short of actually writing code. In many such questions, a piece of "skeleton" code is presented, with one or more lines missing. The desired function of the code is specified. The options of the multiple-choice question then provide candidates for the missing lines of code. The incorrect options are designed around the errors that students frequently make, such as confusing the contents of an array element (eg. "x[i]") with the position of an array element (eg. just "i").

Some of the questions take another approach, requiring the student to identify the behaviour of the code. Below is an example of such a multiple-choice question.

The code for a class called "Counter" is given as follows (consider carefully the use of the "static" keyword).

```
public class Counter
{
        private static int count = 0;

        public void increment( )
        {
                ++count;
        }

        public int getCount( )
        {
                return count;
        }
}
```

If the following code is executed:

```
  Counter counter1 = new Counter( );
  Counter counter2 = new Counter( );
  counter1.increment( );
  counter2.increment( );
  int total = counter1.getCount( ) + counter2.getCount( );
```

the value in "total" is:

a) 0
b) 1
c) 2
d) 3
e) 4

The multiple-choice exam contributes a possible maximum of 70 marks towards a student's final mark. In both semesters when we have run this criterion-referenced assessment scheme, we have set a two hour exam containing 30 multiple choice questions.

In their multi-institution study, McCracken *et al*. (2001) noted that some weak students merely aim to obtain a program that compiles cleanly, and after achieving clean compile, "*they are then surprised by what the program really does when presented with data"*. The traditional emphasis on writing code is frequently accompanied by marking schemes that unduly reward clean compilation at the expense of working code. The style of multiple-choice question we use leaves students with little alternative but to read and understand a piece of code. Such testing at the "knowledge" and "comprehension" levels better prepares students for the time when they will eventually write - and test - their own code.

2.4 Teaching Issues

Our emphasis on reading leads to a clearer and less discursive lecturing style, where we simply read though and discuss complete class definitions. Since "P" students are not required to write original code, there isn't the need to move quickly from a single example to a deep general principal, where the principle is illustrated by several unrelated small snippets of context free code. In a typical lecture, we simply discuss concepts as they arise in the (carefully chosen) complete programs that we are reading. (The stronger students discover the variations on their own, through active exploration.)

2.5 Discussion

An examination of student performance on each multiple-choice question indicates that students are not passing by rote learning. The exam for our second running of the subject contained a mix of questions. Some questions had been seen by students before the exam, and some questions were "unseen" (ie. were about code that students had not seen before the exam). As one would expect, students who failed the exam (ie. scored less than 70%) performed better on the previously seen questions than on the unseen questions.

An examination of 27 "bottom passing" students, who scored 70-77% in the most recent exam, shows only weak evidence of rote learning. There were some unseen questions that fewer than half the passing students answered correctly. The "Counter" question shown earlier is one of the unseen questions that bottom passing students did poorly, with only 7% of them getting it right. However, poor performance on this "Counter" question can be put down to a poor grasp of "static" among bottom passing students. (Note that the "Counter" question even tells the student "*consider carefully the use of the "static" keyword*", so the students did not simply miss that detail.)

Ignoring that minority of poorly done unseen questions like "Counter", typical performance on seen and unseen questions was comparable. On the majority of unseen questions, 63-85% of passing students correctly answered each of these questions, with a median of 78%. On previously seen questions, 63-89% of passing students correctly answered each of these questions, with a median of 81%. These comparable percentages suggest that bottom passing students do not rely on rote learning.

3 Application/Analysis and Grade "C" / "D"

Bloom (1956) distinguishes between comprehension and application as follows: *A demonstration of "comprehension" shows that the student can use the abstraction when its use is specified. A demonstration of "application" shows that he will use it correctly, given an appropriate situation in which no mode of solution is specified (p. 120)*. For example, if a student is told to write a statement so that a given object is registered as an action listener for a given button, then that is an exercise at the "comprehension" level. If, however, a student is instructed to implement a specific graphical user interface for one or more objects, and it is left to the student to realise that, among other things, listeners must be registered with the appropriate GUI objects, then that is an exercise at the "application" level.

An "application" level exercise is more limited than an open-ended "synthesis" exercise. Bloom (1956) states that "application" is the *use of abstractions in particular and concrete situations (p. 205).*

The above considerations led us to formulate the following set of objectives for the "C" and "D" grades:

If you successfully complete this subject to the level of credit (mark 65-74) or distinction (mark 75-84), you should satisfy the objectives of a pass and also have demonstrated, within a small well-defined program context, that you are able to:

(a) Apply the basic OO programming concepts of classes, instances, events, and methods.

(b) Apply the basic program control constructs of sequence, selection and iterations.

(c) Take an informal problem description and translate it into a readable, working program.

(d) Apply the basics of testing and debugging.

We believe these objectives can be assessed in a single, traditional programming assignment, provided the code to be written by students is not too large, and is reasonably well specified. We advocate that students be first given a working program containing several classes. Our assignments assume the students have already mastered the material in the lab exercises. The students are given the assignment at about mid-semester, and have 3-4 weeks to complete it.

Our approach to developing assignments is supported in the literature. We design assignments that are broadly consistent with the eight BlueJ assignment development guidelines of Kolling and Rosenberg (2001). Experimental research by Applin (2001) suggests students learn programming better by extending well-written, well-documented code.

In the most recent semester's assignment, students were given an extensive quantity of code, in twelve classes. The assignment was broken into six tasks, of increasing difficulty. Each of the first five tasks required the students to generate small amounts of code; usually about 10 lines of code, and no more than about 15 lines. The learning emphasis was not on writing code, instead the emphasis was on reading the provided code, and coming to understand how the code the student was to write must interact with the code provided. Only the sixth and final task required extensive code, when students were required to add an entire class.

The six assignment tasks were given equal weight in marking, and provide a possible 16 marks toward a student's final mark.

(A student can be awarded a "C', which is a mark of at least 65-74, on the basis of the multiple-choice exam alone, without attempting the assignment. To do so, however, a student needs to answer correctly 28 out of 30 questions in the multiple-choice exam. Our experience has been that, while a few students do score that highly in the multiple-choice exam, most have also attempted the assignment. In the first semester when we ran this assessment scheme, one student in a class of almost three hundred students gained a credit without attempting the assignment. We were content with awarding a "C".)

Since the "P" grade is entirely determined by the lab exercises, the lab exam, and the multiple-choice exam, participation in the assignment does not directly affect a student's likelihood of failure. In the first semester when we ran this criterion-referenced assessment scheme, approximately one third of the class elected to settle for a "P" and not attempt any part of the assignment. Among those who did attempt the assignment, we found the atmosphere less fraught than in traditional classes, where the assignment does influence pass/fail. This calmer atmosphere is more conducive to learning and less conducive to plagiarism.

3.1 Analysis

Thus far in our discussion of the "C" and "D" grades, we have only discussed the "application" layer of Bloom's taxonomy, and ignored the "analysis" layer. That is because, at this stage of the development of our subject, we have not made extensive use of "analysis".

Bloom states that "analysis" *emphasises the breakdown of the material into its constituent parts and detection of the relationships of the parts and the way they are organised (p. 144).* There is rich scope in object-oriented programming for students to explore the relationships between classes, and to be assessed on their ability to describe those relationships. Our existing approach to the assignment already requires "analysis", to some extent, since we provide extensive code and students must make sense of it, but there is much greater scope for the exploration of "analysis". Perhaps in the future, while further exploring "analysis", a natural set of criteria will be found for distinguishing a "C" student from a "D" student.

4 Synthesis/Evaluation and Grade "HD"

Before describing what we want to reward with an "HD", we describe what we do not want to reward. We do not want to reward (1) long hours of mundane work, and (2) prior knowledge of programming. Also, while discussion of clarity in code is certainly pursued in lectures, our subject does not explicitly attempt to teach and assess design principles, so it would be inappropriate to explicitly reward good design in marking schemes.

We have observed that, when teaching under a more traditional assessment regime, students have not absorbed our exhortations about the importance of good, clear code. We have come to the view that the importance of good coding style is best learnt by experiencing poor style. (It would however, be counter-productive for we teachers to provide the bad examples!) Consequently, our aim is to demonstrate the importance of good code to our best students by getting them to suffer a little (1) struggling with their own code, and (2) reading each other's (bad!) code. Since the subject that follows our subject emphasises design, we would assert that the experience our "HD" students have in this project motivates them for the subsequent subject. Jenkins (2001) noted that some students are motivated extrinsically (eg. a well-paid career) whereas other students are motivated by an intrinsic interest in the subject for its own sake. In providing our best students with the experience of bad code, we hope to cultivate an intrinsic interest in good coding and design.

Bloom has defined synthesis as "*Skill in writing, using an excellent organisation of ideas and statements*" (p.169). He was referring to essay writing, but his definition fits programming well. He further states that synthesis is the "*ability to write creatively a story, essay, or verse for pleasure, or for the entertainment or information of others*" (p. 169). Over time, perhaps the 'pleasure' and 'entertainment' in learning to program has been lost. In our subject, we seek to give the stronger students the opportunity to play.

The above considerations lead us to the following objectives for the "HD":

If you successfully complete this subject to the level of high distinction (mark 85-100), you should satisfy the objectives of the credit/distinction level and will have demonstrated, within a larger and less well defined context, that you are able to:

(a) Synthesise and evaluate the basic OO programming concepts of classes, instances, events, and methods.

(b) Synthesise and evaluate the basic program control constructs of sequence, selection, and iteration.

(c) Develop your own informal problem description and translate it into a readable, working program.

(d) Synthesise and evaluate GUI design, from both the user and programmer perspective.

(A student can in theory be awarded a "HD', which is a mark of at least 85, without attempting the assessment tasks described below, since the multiple-choice exam and the assignment provide a maximum possible 86 marks. On the two occasions that we have used this assessment scheme, no student has achieved that feat.)

4.1 Synthesis: Individual Project

For a "HD", students are first required to write a program of their own choice. The following list of criteria is an edited version of the criteria given to aspiring "HD" students:

- Your program must have at least 4 classes that you have written entirely on your own, containing at least three methods other than get/set methods. (You may use code from other sources, but this code should be kept in separate classes from your own code) You should clearly distinguish between the classes written by you and the other classes.
- The code written by you must total to at least 100 lines (excluding comments and the usual things omitted by industry standards for measuring lines of code). A "HD" is not a reward for writing large quantities of poor code. The primary requirement is for quality code that meets the minimum number of lines criteria.
- Your program must have a graphical user interface. The interface must be more sophisticated than that of <a simple example given in lectures>. The interface should exhibit good design attributes as discussed in the lecture on GUI design. The quality of the interface will be an important consideration in marking. The GUI should be implemented in AWT. Do not use Swing. You must craft the GUI yourself. Do NOT use the GUI wizard of a development environment.
- The project must not be yet another implementation of a common Java programming exercise, such as a calculator. It must not be an implementation of an exercise from the textbook. You are expected to demonstrate some initiative, and create a project that is your own idea. If you are in doubt as to whether your idea is worthy, discuss it with <the lecturing staff>. (Tutors may have an opinion on your HD project, but they are not directly involved in the marking of HD projects.)
- The code must be written in good style. For example, it must be well indented, well commented, and variable names must be well chosen. The style in the textbook is acceptable.
- It must contain loops and "if" statements.
- The code must compile and run. Also, evidence of a coherent and comprehensive test strategy must be submitted.
- The submitted code must compile and run within BlueJ. (However, you may use a non-BlueJ environment to develop your project.) A submission that does not compile and run in BlueJ is likely to receive a zero mark.
- The ReadMe.txt file that is provided in the BlueJ environment must contain an explanation of how to run the program.

- Known bugs must be reported.
- Personal Software Process documentation must be supplied. (This requires the students to effectively keep a diary of their project's development. The diary helps students reflect on how they work. It also makes plagiarism more difficult, since a diary would have to be credibly faked.)
- Students are warned that submissions will be distributed among other students via the web. Thus, the submitted code should not contain your name, or your student number.
- If the project requires data to be entered for testing, then test data should be automatically loaded at startup. Projects should not interface to a database system. A submission that does not run because a datafile is missing from the submission package is likely to receive a zero mark.
- The above criteria do not guarantee that your program will be judged suitable for a HD. The above are minimal essential criteria.

Students are warned that a project unworthy of a "HD" will probably be awarded zero marks, thus discouraging students who might cynically submit a poor effort in pursuit of a few marks to enhance the chances of getting a lower grade.

4.2 Evaluation: Student Peer Review

The project described above is merely a student's "entry ticket" into the "HD community". After completing their own projects, students must then examine the projects of two other students against a set of criteria designed to evaluate the program text and the program operation.

The following is an edited set of evaluation criteria given to students:

- Is the documentation clearly written?
- Is the code written so as to make it easy to understand?
- Is the program simple and easy to start?
- Is the user interface designed to work for people other than the programmer?
- Did it work for you?
- Are all errors found by you documented?

The evaluation criteria are deliberately vague. We want our prospective "HD" students to reflect. We deliberately do not give them a check list.

4.3 Discussion

The question may well be asked; how can students who have not been explicitly taught design principles be expected to perform an evaluation? The premise of the question is mistaken: we believe that students will be better motivated to learn about good design in future subjects having been exposed to poor design. We want to decrease the emphasis on writing. We want our strong students to think of a good program as something that is worth reading. (Having said that, however, we have found that the "HD" student exhibits good design intuition. For example, they are able to detect artificial divisions of functionality between classes that were made by another student to raise the number of classes to the four required.)

In the first semester that we ran the HD project and critique, about 10% of the class attempted the project. As Table 1 testifies, only 6% of the class succeeded in achieving a "HD". In our faculty, it is thought that 10% of students in any subject should get the highest grade, so we may need to fine tune our "HD" criteria. (Such a fine-tuning is not at odds with the principles of criterion-referencing. It is appropriate to design criteria with certain grade distributions in mind. The essential principle is to apply the published criteria even when the resultant grade distribution is not ideal.)

Some readers might wonder why we ruled out the use of Swing, and preferred AWT. One minor consideration was marking parity. When marking projects, how were we to fairly mark a mix of GUIs developed with both Swing and AWT? More importantly, AWT and not Swing was taught in lectures, and we did not want to force a student to learn Swing just so they could evaluate another student's Swing-based GUI. Perhaps most important was the concept of enforcing the spirit of a community. A standard, which did not satisfy all members of the community, was enforced so that all members of the community could share code: does not that the same thing happen in any software shop? We wanted to encourage a sense of community. We want to discourage the notion of the lone hacker.

Some readers might have been surprised to see that students were not required to use arrays in their "HD" project. This decision was made for two reasons. First, it allows students to start the "HD" project earlier in semester. Otherwise, students without prior knowledge of programming would have been disadvantaged. As stated earlier, we did not want to simple reward prior knowledge of programming. Second, we believe there is a great deal to Object-Oriented programming other than arrays, particularly when a GUI is required, so it simply is not necessary to insist that students use arrays.

5 Student Feedback

In the second semester that we ran this criterion-referenced assessment scheme, the class was surveyed as a routine part of the faculty's quality assurance process on all subjects. The students were asked the faculty's standard set of questions, to which they answered, on the common 5-point Likert scale (strongly disagree, disagree, neutral, agree, strongly agree). The survey was done before the assignment and HD projects were submitted, (and also before the multiple choice exam, which is held during the exam period at the end of semester). Consequently, some students were determinedly neutral in their responses to the survey.

One of the survey items was "The subject was delivered in a way which was consistent with its stated objectives". In response 22% of the class strongly agreed, 44% agreed (total 66%), 33% were neutral, and none disagreed. We take this as an indication that none of students rejected the assessment scheme in principle, and many students liked it.

Another survey item was "My learning experiences in this subject were interesting and thought provoking". In response 11% of the class strongly agreed, 58% agreed (total 69%), 18% were neutral, and 13% disagreed. One of concerns had been that the 37% of the class who eventually achieved a "P" may have been bored by an assessment diet of lab exercises, lab exam, and multiple choice exam, but this does not appear to be the case.

A survey item of great interest was "I found the assessment fair and reasonable". In response 7% of the class strongly agreed, 57% agreed (total 64%), 30% were neutral, and 7% disagreed.

6 Beyond the First Course

Bloom wrote, *"It is quite possible that the evaluative process will in some cases be the prelude to the acquisition of new knowledge, a new attempt at comprehension or application, or a new analysis and synthesis"* (p.185).

Having graduated from our introductory programming subject, what should students go on to do in their second subject? We are not yet teaching such a course, but we have some initial thoughts on how to apply Bloom's taxonomy again, to develop criteria for each grade.

For a grade of "P" in the second subject, students should be able to write modest but complete programs. They should also exhibit "knowledge" and "comprehension" of issues in good program design, but without always manifesting that knowledge when writing their own modest programs.

For a grade of "C" or "D', students should exhibit "application" and "analysis" level performance at program design. In their assignment work, we would expect these students to manifest a working grasp of design issues. There is extensive scope for "analysis" tasks, where students would identify the parts of well-constructed software.

For a grade of "HD", students should would once again exhibit performance at the "synthesis" and "evaluation" levels, would once again conceive and build their own project, then critique the projects of other students. This time, however, a higher standard would be expected, since the students would now be aware of good software design principles.

Beyond the second course, we see a sequence of subjects comprising the software-engineering strand of an IT degree. Implicit in the grading schemes of many such strands is the view that the P-average student should emerge with strong programming skills, and the potential to play an important role in a software development project. If that is the case, then what is the role of the higher achieving students? As we think through the full consequences of our approach to grading, we find ourselves thinking that only consistently high achieving students should be considered as having demonstrated the potential to eventually play, after appropriate industrial experience, a leading role in serious applications design and development. The solid "C" student might work under close supervision on such projects. A student who graduates from a software-engineering strand with a string of "P" grades should not be involved in serious applications development. The writing of small non-critical programs may be part of such a person's job, and they may be required to communicate with software developers, but that should be the limit to their involvement in software engineering. At the heart of our teaching problems, is the mistaken belief that even P-average students should be taught all the skills required to write software for nuclear power stations.

7 Conclusion

Given the goal of maximising the potential of each student in a disparate class, it is hard to see how that goal can be achieved when the same task, or even broadly the same task, is given to all students. The strongest students will not be challenged, while the weakest students will flounder.

The message of Bloom's taxonomy has always been that we need a mix of strategies, to test students at all levels of the taxonomy. Decades ago, when Bloom's taxonomy was first published, the effect of the taxonomy was to highlight that school teachers placed too much emphasis on testing knowledge and comprehension. Today, the taxonomy highlights that IT academics place premature emphasis on the higher levels of the taxonomy.

Our criterion-referenced grading scheme brings together several ideas – lab exercises, lab exams, multiple choice exams, assignments, projects, and peer review - all of which have been used before. Our contribution has been to bring these disparate grading techniques together, uniting them in a coherent explicit grading philosophy based on Bloom's taxonomy. The grade distribution for our first running of this approach might indicate that some ongoing fine-tuning is required, but the basic approach appears to be sound.

Most IT educators will have encountered a brief summary of Bloom's taxonomy, but we encourage all IT educators to go back to the primary source, and read the entire 200 page document. There is a sophistication to Bloom's arguments that is lost in those summaries.

8 References

APPLIN, A.G. (2001): Second Language Acquisition and CS1: Is * == **? *Proc. SIGCSE Technical Symposium on Computer Science Education*, Charlotte NC, USA, 174-178, ACM Press.

BLOOM, B.S., et al. (1956) *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*, Longmans, Green and Company.

BUCK, D., and STUCKI, D. (2001): JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum. *Proc. SIGCSE Technical Symposium on Computer Science Education*, Charlotte NC, USA, 16-20, ACM Press.

DEIMEL, L. and NAVEDA, J. F. (1990) *Reading Computer Programs: Instructor's Guide and Exercises,* CMU/SEI-90-EM-3, Carnegie Mellon University. http://www.sei.cmu.edu/publications/documents/ems/90.em.003.html (October 13, 2002).

FINCHER, S., (1999) What are We Doing When We Teach Programming? *Proc. Frontiers in Education '99,* 12a4-1 to12a4-5*,* IEEE Press.

JENKINS, T. (2001): The Motivation of Students of Programming. *SIGCSE Bulletin,* **33**(3): 53-56.

KERNIGHAN, B., and PLAUGER, P (1981). *Software Tools in Pascal*. Addison-Wesley.

KOLLING, M., and ROSENBERG, J. (2001) Guidelines for Teaching Object Orientation with Java. *SIGCSE Bulletin,* **33**(3): 33-36.

LISTER, R., (2000), On Blooming First Year Programming, and its Blooming Assessment. *Proc. Fourth Australasian Computing Education Conference* (ACE2000), Melbourne. pp 158-162.

LISTER, R., (2001) Objectives and Objective Assessment in CS1. *Proc. SIGCSE Technical Symposium on Computer Science Education*, Charlotte NC, USA, 292-296, ACM Press.

McCRACKEN, M. (2001) A multi-national, multi-institutional study of assessment of programming skills of first-year CS students, *SIGCSE Bulletin,* **33**(4):1-16, ACM Press.

ROUMANI, H., (2002) Design Guidelines for the Lab Component of Objects-First CS1. *Proc. SIGCSE Technical Symposium on Computer Science Education*, Cincinatti, KT, USA, 222-226, ACM Press.