Holistic assessment criteria – Applying SOLO to programming projects

Errol Thompson

Department of Information Systems
Massey University
PO Box 756, Wellington 6140, New Zealand

E.L.Thompson@massey.ac.nz

Abstract

How you define your assessment criteria should influence the way the students approach the assignment. Does this mean that if we use a holistic criterion-based assessment strategy that students will look more holistically at the topic rather than focussing on the pieces for which they think they can gain satisfactory marks? A holistic set of assessment criteria for programming assignment work based on the SOLO taxonomy is presented, and reflections on the use of this approach over three years are discussed.

Keywords: Assessment, programming, criterion-based assessment.

1 Introduction

The initial moves to a holistic approach to grading for programming assignment work was reported in Thompson (2004) where the holistic approach was compared with a scoring / weighting rubric (Maki, 2004). The institutions, in which this work was carried out, used criterion-based strategies for essay and report writing assignments that endeavoured to assign higher grades for critical thinking. There was difficulty translating these to programming assignment work where the emphasis seemed to be on satisfying the required functionality according to a set of predefined programming standards.

The SOLO taxonomy (Biggs and Collis, 1982) provided a solution. In his book on improving learning in the university context, Biggs (1999) provided an example of how he applied the taxonomy to an essay style assessment. This sparked a number of trials of different SOLO based grading criteria for both essays and programming exercises in the context of object-oriented software development. These trials were completed with second and third year papers.

Biggs (1999) focuses extensively on the use of the SOLO taxonomy in university education. He contends that the strategy led to essays that integrated knowledge rather than simply focussing on a single aspect or providing a shopping list of concepts relevant to the topic. Biggs' criteria seemed to provide a better criteria than the concepts of width, depth, and distance that had been used for assessing learning journals (Thompson, 1998, Thompson, 1997, November, 1996, November, 1997). Using learning journals in a distance education course revealed that some students were capable of integrating knowledge while others simply rewrote material from references or notes with limited reflection or integration.

In examining programming assignments, it was possible to observe the shopping list style answer approach to writing code. The criteria for these assignments didn't encourage code reuse or have any emphasis on the structure of the code. The program needed to be syntactically correct, implement a required set of functionality, and utilise reasonable programming practices. Although the marking criteria included the ability to use modules (i.e. subroutines and functions), this didn't lead to the students integrating code or reducing code duplication. Modules tended to be large and perform multiple tasks. The use of documented testing strategies was also given little importance by the students with often buggy and incomplete code for all functionality being handed in for marking. Students appeared to see programming as an exercise in completing as much of the required valid data path functionality without concern for the overall integrity or quality of the product.

The criteria described in this paper attempts to address these issues by drawing on the concepts underlying the SOLO taxonomy. Other experience with using the SOLO taxonomy in evaluating responses to program reading questions of novice programmers is documented in Whalley et al. (2006), Lister et al (2006), and Thompson et al. (2006). In these papers, the SOLO taxonomy was selected to analyse the data gathered from a series of programming related questions. These questions were not written with the SOLO Taxonomy in mind.

This paper reports on the use of SOLO to define a set of criteria that are given to the students to influence their approach to the programming task and to assess the work that the students present for marking. Biggs (1999) and Hattie and Purdue (1998) describe this type of usage.

2 Methodology

The holistic marking criteria were originally introduced to papers in 2002 taught. They have been used for essay,

design and programming assignments. These papers have been at all levels within a degree program.

The lecturer involved initially evaluated the marks obtained against the previous marking strategies used for these papers (Thompson, 2004). In that paper, it was shown how the strategy had limited impact on the grade of a good student (A+) but could cause a lower grade student (C - C+) to fail unless they changed their approach to the assessment. The lecturer concerned argues that this is precisely the outcome that was wanted from the holistic criteria.

This paper endeavours to use qualitative data drawn from the lecturer's journal to identify whether the students' approach to the assignment was changed as a result of using the assessment strategy? The lecturer recorded key questions raised by students about the assignments and marking strategies. These were recorded following the lecture or laboratory sessions.

3 SOLO categories

The SOLO taxonomy (Biggs and Collis, 1982) provides a series of categories based on the structural relationship of the material being presented. The categories of the SOLO taxonomy are defined as:

Category	Description
Prestructural	a) Misses the point of the exercise or plagiarises from other material. b) The presented information has little or no relevance to the requested requirements.
Unistructural	a) Focus on one conceptual issue or naming things. b) Shows minimal understanding by only giving serious consideration to one feature or requirement.
Multistructural	a) List of items but no relationship between items. b) The emphasis here is on "knowledge-telling" (i.e. look at how much I know).
Relational	a) Shows understanding through integrating concepts and ideas. b) Understands how to apply the concept to a familiar problem.
Extended Abstract	a) Relates an existing concept or principle in such a way that they are able to handle unseen problems. b) Questioning and going beyond existing principles

Table 1: SOLO categories

The prestructural category includes the criterion that the essay or code has to meet a minimum standard of presentation. This doesn't mean the assessor is looking for all the grammatical or spelling errors in an essay. An essay has to be able to be read in reasonable time without the reader being caught by obvious grammatical problems. There are tools that the student should learn to use to help them verify the grammar or spelling. The issue is whether the information is presented in a way that makes sense to the reader. If grammar or spelling gets in the way then the student hasn't applied the basic tools that are available.

4 Programming SOLO categories

This section outlines the criteria based on those used in a second year programming paper that emphasised test-driven development and refactoring. The assignment was provided as three iterations over the twelve week period of the course. Each iteration added another feature that reused some of the existing functionality either explicitly, such as validation rules for input data, or implicitly through the need to use closely related processing structures or techniques such as access to a database. The iterations also tried to introduce a requirement for different programming constructs or techniques.

The objective of the paper was to introduce the students to the techniques of test-driven development and refactoring. They all had passed previous introductory and intermediate level programming papers. Limited emphasis was placed on learning new language or framework features.

In the criteria for the assignment, some introductory notes were included to emphasise the different focus used in the assessment criteria. These notes were:

1) Exceeding the minimum requirement in one area for a grade will not see a higher grade awarded. You must show that you are applying all the principles consistently to be awarded a higher grade.

2) Features in the following criteria relates to all aspects of the assignment. That is, the programs required functionality, the design of the user interface, and the implementation of an automated testing strategy.

3) Programming standards include the use of good code layout, variable names, and the elimination of code duplication. The code should be readable with the minimum of internal comments. That is, it should be self documenting.

4) Form design should endeavour to be consistent with the conventions of windows based applications.

5) The tab sequence on the form should follow a logical pattern.

The first of these notes was intended to discourage the students focussing on implementing all the functional requirements and ignoring the requirement to use test-driven development and refactoring. Simply completing more functionality would not gain a better mark if the required practices where ignored. This was further emphasised by the second note.

As well as the program code, the students were asked to provide a document that described the reasoning for their design. They were not encouraged to produce external detailed design documentation but were encouraged to use coding standards that promoted readability and self-documentation of code. The emphasis on internal documentation was placed on comments that would help explain why a particular programming approach was used rather than a simple description of what the code was doing. Comments that simply stated the obvious (i.e. adds the two values together) were discouraged.

4.1 Base Standard

It was also stated in the assignment brief that programs that did not compile or run would be returned without being marked. It was not the marker's task to fix such problems. As a programming paper, the students had access to a compiler to validate syntax and were being encouraged to focus on incremental development rather than attempting all functionality and getting none of it working. There should be no obvious faults in the submitting code. The emphasis in the assignment instructions is to have completed and tested features rather than to have started all the features but have few of them completed.

4.2 Inadequate work (Prestructural)

This category could also be defined as absolute fail (E grade) in terms of the assigned grade. In line with the SOLO taxonomy, a student graded into this category was showing that they did not understand the task or what was expected of them.

4.2.1 Issues

The primary focus of this category was that the student either showed inadequate knowledge or had completed an inadequate amount of work. In line with the concepts of plagiarism in essay writing, this category included simply copying example code. Such copying would normally produce a program that failed to deliver the working functionality unless the plagiarism involved copying another student's code. During the lectures, code examples were given out that illustrated a range of solutions to particular types of coding problems and had been used to discuss good coding practice and possible design options for different aspects of coding business applications. At no point were the students given a full system solution so any copying of solutions would involve the selection of sample code that would not adequately combine into a solution.

As well as ruling out copying, these criteria specified the minimum standard required to be considered as making a serious attempt at the programming exercise. The target of 30% represented slightly less than one iteration for the assignment. Inadequate progress was seen as being an indicator of an inability to tackle the programming tasks.

A minimum standard was also presented for coding, user interface design, and application structure. If they were given minimum consideration by the student then the assignment submission was dismissed as unacceptable.

4.2.2 Criteria

The inadequate work (prestructural) criteria were defined as:

Copying code or *no understanding* of programming issues.

- Application attempts to copy example code with minimal changes
- Application is unrelated to requirements
- Application delivers less than 30% of the required functionality.
- No attempt has been made to apply programming or user interface design standards or good application structures.

4.3 Single aspect (Unistructural)

This category was considered a marginal fail (D grade). The student showed some understanding but was operating an inadequate level to be able to participate in a programming environment.

4.3.1 Issues

The reason for a student's work to be graded in this category was that they had focussed on one aspect of the assessment. This might have been that one feature of the system had been implemented or that one aspect of the required implementation tasks was utilised. For example a student may have concentrated on the design of the user and ignored the implementation of processing logic or a testing strategy. If a student completed all the functionality according to the specifications but wrote no automated tests or documented no testing strategy then this was regarded as focusing on a single aspect of the assignment task.

In attempting to define this category, two dimensions were taken into account. These were the features or scope of the system and the range of programming techniques or constructs that should be used. Focussing on only one iteration or feature set of the system was regarded as a single aspect. Likewise, focussing on a single programming technique or construct was regarded as a single aspect.

Because of the way that the iterations where defined, students who only completed one iteration or who were only beginning the second iteration were unlikely to demonstrate anything other than a single aspect approach to the assignment.

4.3.2 Criteria

The single aspect (unistructural) criteria were defined as:

Shows a *limited understanding* of programming and application development issues. Parts of features are implemented without ensuring successful operation. Some of the issues considered include:

- Application partially operates with significant obvious problems.
- Application delivers 30-50% of required features as specified in the requirements.
- Programming standards, application structures, and user interface design standards are not applied consistently.

4.4 Disjoint project (Multistructural)

The greatest grade range was assigned to the disjoint project (multistructural) category. In defining this category, the grade band was split into two categories.

The lower band of this category represented those students who were only just making the standard in more than one aspect of the project. They were assigned a grade that would give them a marginal pass for the assessment (C grade). These students might be able to participate in projects where they can focus on a specific aspect and not have to deal with all aspects of a project.

In contrast those in the higher band were endeavouring to satisfy the standards in most aspects of the project. However, they were not seeing these aspects as related or integrated. Each was handled as though independent of each other. Many of these students would make good journeymen on programming projects where they could follow the lead of others. These students were assigned a adequate pass grade (B grade).

4.4.1 Lower disjoint project band issues

The lower disjoint project band focussed on the use of a limited range of concepts and techniques or on implementing limited functionality. If a student attempted two iterations in a manner where each iteration was considered as single aspects then their mark would fall into this lower category band. An attempt to complete more than one iteration was needed for the student to be graded above this band. One iteration if completed fully utilising all the required programming techniques and with automated test would be at the bottom end of this grade band. The issues of integration of code between iteration requirements would not be illustrated in the student's solution.

This lower category might also be represented in a project where the student had completed the full scope of the project but only partially addressed the testing requirements or user interface design issues.

4.4.2 Lower band criteria

The lower disjoint project (multistructural) criteria were defined as:

Is able to *complete a working* piece of code to a base standard. The completed features are implemented but without ensuring consistency in operation or implementation. Some of the issues considered include:

- Application operates without obvious problems (i.e. does not crash when executed).
- Application delivers 50 to 60% of required features as specified in the requirements.
- Application inconsistently applies programming standards and user interface design standards.

4.4.3 Upper band issues

In contrast to the lower disjoint project category band, the higher disjoint project category recognised that a large amount of the system functionality might be implemented but that there was limited or no integration of that functionality. At least two iterations of the assessment project would need to be attempted to be considered for a grade in this band. The project may have attempted all iterations but each iteration appeared in the presented work as if it was a totally independent programming project or included a high level of code duplication because the student had not seen the commonality that was possible in developing the solution. The required system specifications might have been fully met but the user has what appeared to be three totally independent projects.

4.4.4 Upper band criteria

The higher disjoint project (multistructural) category was defined as:

Is able to *complete a working* piece of code to a base standard. The completed features are implemented as if they don't belong together. Some of the issues considered include:

- Application operates without obvious problems (i.e. does not crash when executed).
- Application delivers 60 to 70% of required features as specified in the requirements.
- Application consistently applies programming standards and user interface design standards.

4.5 Unified project (Relational)

This is the level that we wanted most students to operate at. These students were assigned a grade that indicated their high level of competency as a programmer (A grade). At this level, they were seeing the relationships in what they were attempting to achieve although not stretching beyond their current knowledge. It would be expected that students operating at this level would be able to deal with most programming tasks that utilised familiar techniques or practices. They might struggle where novel solutions were required.

4.5.1 Issues

In this category, the student has completed the assessment work to a level that shows integration of the iterations and utilisation in a coordinated manner of a wide range of programming techniques and constructs for the intended purpose. Two iterations of the project needed to be completed and a portion of the third iteration attempted in order for the student to be considered in this grade band.

At this level, the student has identified the commonality in the functionality of the code and has refactored the code to eliminate duplication. The student has also recognised the need to apply all the programming techniques consistently to achieve the projects objectives.

4.5.2 Criteria

The unified project (relational) criteria were defined as:

Is able to *apply* the programming concepts taught and consistently *uphold* the standards and structures from example code. The completed features are implemented in a way that demonstrates the integration of ideas and of the system being developed. As above plus:

- Application delivers over 70% of the required functionality as specified in the requirements.
- Application structure is clean and matches standards.
- Application is documented externally to ease understanding.
- Minimal duplication of code and good reuse between different functional components.

4.6 Outstanding work (Extended Abstract)

This category was reserved for those students who had exceeded the expectations for the assessment. These were the possible innovators and ideas people of a project. As such they were assigned a grade that reflects excellence (A+ grade).

4.6.1 Issues

At the lowest end of this category would be the student who has completed all the iterations with fully integrated code and utilising the full range of programming techniques and constructs. Ideally, they should have used some constructs that were not explicitly taught in the paper. The focus here was on going beyond just doing the basics. The last iteration provided the students with the opportunity to extend their learning if they wished and to use some alternative techniques not explicitly covered in the paper but which they should have been able to learn based on the learning foundation given in the paper.

The outstanding student in this category would be using additional programming techniques and constructs, and have demonstrated an ability to argue for their inclusion in the project. The student was not expected to add functionality beyond what was requested but to show that they had evaluated and explored new techniques and constructs without formal instruction or guidance. They may also have demonstrated an ability to identify weaknesses in the requirements or design specification.

4.6.2 Criteria

The outstanding work (extended abstract) criteria were defined as:

Shows initiative to experiment with new ideas and is able to *present a meaningful argument* for a revised approach. Achieves what is specified in an integrated way and addresses issues beyond those clearly stated in the assignment. As above plus …

- Application delivers over 80% of the required functionality.
- Application design shows integration of task components.
- Utilises programming techniques and constructs that were not explicitly taught in the paper
- Documents reasoning for choice of approach to application design and coding.
- Documents task integration issues.

4.7 General comment

The objective in writing the criteria presented here was to give the students a clear indication of the intent for each of the marking criteria and to provide some basis for them to check whether they had achieved a specific standard. Those operating at the outstanding work category level didn't need a checklist but those at the lower levels often seemed to need explicit checklists of what was expected.

5 Cases

The previous section introduced the concepts of using the SOLO taxonomy for defining the assessment criteria for programming projects. The criteria described are those used in a particular paper. They need to be adapted for each paper. In this section, the application of the criteria is discussed in relation to specific student cases. The described cases are based on experience over the last two years.

5.1 "How can I be sure that I have done enough?"

This first case related to the difference between criteria that defined as a checklist or scoring / weighting rubric (Maki, 2004) and the holistic approach of the SOLO type categories. The students had phrased this as "how many marks will I get if I only do …?" or "how can I be sure that I have done enough to get a .. grade?"

These questions still occurred with scoring / weighting rubrics but seemed to be even more pronounced when the SOLO taxonomy criteria are used. The scoring / weighting rubric provided a form of checklist which the students used to check off whether they thought they had done enough work. When pushed they accept that the SOLO categories had given them this information and that it was simply that it was not in a form that they are used to using.

This did lead to the protest that this was not how they were assessed in other papers. In other papers, they felt they could leave out practices required by the assessment. They contended that the criteria didn't give them flexibility in how they approached the task. To some extent this was true, but in reality they had considerable flexibility in how they created the solution and in the type of tests that they utilised. What they didn't have flexibility in was providing the required functionality and the proof that the functionality was working as required.

Part of the reason for this difficulty was that the SOLO taxonomy criteria were only being used in papers taught by one lecturer. Students were not familiar with this style of criteria or approach to assessment. Despite having talked through the criteria, the style of assessment, and the desire to help them understand professional practice, the students had difficulty understanding or accepting the expectations of the criteria.

The students were indoctrinated with the assessment practices of the department and institutional culture. This indoctrination wasn't simply for the criteria but for the nature of assessment and ability to negotiate changes to due dates, and criteria.

This problem wasn't directly related to the use of the SOLO taxonomy. It applied to the use of any alternative assessment strategy or assessment criteria. To use an alternative assessment strategy or criteria requires making the effort to assist students to understand the expectations.

5.2 "Do I really need to do that?"

This case had some similarities to the previous case except that this wasn't based on issues of inflexibility but more whether something was really needed. In the assignment for which the above criteria where used, the completion of a feature involved both the code and automated tests. This included elements of data validation. Under a conventional marking matrix, students knew that they could ignore the automated tests and still pass the assessment possibly with a very good mark. They also knew that as long as they had the primary processing path working (i.e. valid data) and testing for that path that they would pass in other papers. With the SOLO strategy, completion of functionality meant they needed to be more thorough in their work including dealing with invalid data in order to obtain the same grade. It was necessary to complete the full functionality of a feature and not simply what seemed to deliver the core functionality.

Students in this category either obtained a lower grade than they would have under a scoring / weighting rubric (Thompson, 2004) or they adapted to the new criteria and completed the additional tasks to obtain a similar grade.

5.3 "I need to implement this now"

When a new iteration was handed out there was in the students thinking a need to add new functionality even if the previous functionality was still incomplete and untested. The result often was code that was full of problems and faults. The more functionality they tried to add the more problems and faults occurred often leading to code that failed to run at all. The shift to ensuring the current iteration is complete before moving on had to be continually reinforced and in some cases demanded before help or assistance was given to solving the more difficult problems. This problem was also occurring with conventional scoring / weighting rubrics but students found it easier to ignore the problems and still pass.

This was less of a problem in an offering where individual iterations where taken in and checked for a satisfactory level of completion before the next iteration was given out. The marking strategy handled this situation with minimal adjustment since the student who could not complete the first iteration would not gain a pass grade. Not completing the current iteration put a ceiling on the mark obtained for the project and reinforced the objective of the marking strategy.

When iterations had to reach a certain standard before the student was given the requirements for the next iteration, this led to those students who are struggling to reach the standard protesting that they were being disadvantaged because they were not getting the opportunity to work on the next iteration. These students sometimes obtain copies of the next iteration from a student who had achieved the standard in the belief that if they implemented more functionality they would get a better mark. They failed to recognise the importance of applying key practices or reaching required levels of performance before moving on. They also failed to recognise that failure is part of the learning process. Steve McConnell (2004) says

> "Great designers usually have experience on failed projects and have made a point of learning from their failures. They try out and discard more alternatives. They are often wrong, but they discover and correct their mistakes quickly. They have the tenacity to continue trying alternatives after others give up."

Students who refused to accept that their work is not up to standard were failing to learn the important lessons that would enable them to become better programmers and to gain the freedom that they desired in programming.

A marking strategy based on assigning a portion of marks for each iteration still portrayed the idea that if they didn't meet the required minimum standard for this iteration, they could always pick up marks for the next. The message desired from the SOLO strategy is that each iteration must be completed successfully. This is an attitude that is portrayed in the velocity calculations of agile methods such as extreme programming where a story is not included in the velocity if it is not complete. If the student was not allowed to move forward to the next iteration until they have completed the current to the required level, then they were learning that work needs to be completed before it counts. The velocity count is all or nothing for stories. It isn't an estimate of "I have 50% complete so I am making progress".

5.4 "I have all the functionality there"

This case relates to the students who completed all the iterations as though they were totally independent applications. In this situation all the functionality was there and was fully tested. The problem was that there was no recognition of common features (i.e. repeated validation) and duplication of coding occurred.

In the SOLO taxonomy grading system, this student only received a B grade and some protested that they should be receiving at least an A if not an A+. From a teaching perspective, the B grade was what they should have received because they were not integrating their work but the students failed to see the importance of this integration activity.

It is easy to blame the students for this style of thinking but sometimes it reflects the way that the students have been taught to program. The use of methods came after they were taught all the logic structures. It wasn't taught as part of the process of dividing your code into

functional components or for implementing duplicate logic. Rather it was simply a tool that you used when directed.

A similar attitude exists to the use of objects. Because the student has been taught procedural logic first, they see the problem in procedural terms rather than as interacting objects. When a student is asked why a form is overloaded with logic and why they have not created business objects that contain testable functionality, they respond that it was the way they have learnt to program.

Felleisen et al (2001) and Proulx and Gray (2006) propose teaching strategies that seem to reverse conventional wisdom but lead to students thinking in terms of class and method design ahead of logic design. This approach to teaching may address the issues raised by this case.

5.5 Testing isn't a programmer's responsibility

Test-driven development argues that the automated test should be written before the code is written. This does mean that the programmer needs to have an idea of how the program is to be designed before they write the test. Students took this a step further and argued that they didn't know what the test would be until they had written the code or that it was not possible to write tests for the code because the program could only be tested as a whole or that the programmer wasn't responsible for testing their code.

The result of this attitude was that automated test code was only submitted for a small segment of the code or the provided tests were superficial. Some functionality was implemented but not tested or there were failing tests in the automated test suite.

This was a problem of attitude rather than reality. In order to write the code, the programmer has to have some expectation of what the program will do. If the programmer doesn't know how the code should respond to certain inputs or what should be returned as a result of some coding sequence then they don't understand the requirements. Writing an automated test is the ultimate level of formalisation of the requirements.

> Edsger W Dijkstra (2000) said "A programmer has to be able to demonstrate that his program has the required properties. If this comes as an afterthought, it is all but certain that he won't be able to meet this obligation: only if he allows this obligation to influence his design, there is hope that he can meet it. Pure a posteriori verification denies you that wholesome influence and is therefore putting the cart before the horse..."

Being able to develop a testing strategy based on the requirements shows a clear understanding of what is required. It is only at this point that the programmer has a clear idea of what is required and has, as Dijkstra said, allowed the obligation to "to demonstrate that his program has these properties" to influence the design. Simply gathering numerical proof as evidence of correct operation denies this influence.

> McBreen (2001) argues that "Software craftsmen have a real interest in automated testing because of their investment in their reputations."

Should the grade assigned to the student reflect an "investment in their reputation" or should it be an indicator of a good attempt even if they failed to prove the integrity of their code? Students who failed to accept this message received a lower grade than they would have using a scoring / weighting rubric (Thompson, 2004)

This complaint wasn't a failure of the SOLO taxonomy marking criteria but rather failure of our teaching strategy.

6 Conclusion

This paper outlines criteria that have be used for programming assignments. The cases described show issues that arose and how these issues came from perspectives that were challenged by the criteria.

Did the strategy cause the students to change to a more holistic approach to their assignment work? Clearly there was resistance to change and some students refused to change (cases 5.2 and 5.3). Some of this resistance was caused by lack of familiarity with the approach to assessment reflected in the criteria (case 5.1). Other resistance came from a feeling that they could not select what they wanted to focus on (case 5.2). This reflected a resistance to a holistic approach to the task. Further resistance came from issues of understanding the process of software development and what it meant to have completed an iteration (case 5.3). This wasn't necessarily resistance to a holistic assessment approach to the task. In two of the cases, the resistance came from perspectives of what it means to complete functionality (case 5.4) or to ensuring that code worked as required (case 5.5). Although this might be seen as resistance to a holistic approach to the task, the teacher considered the approach to teaching as a possible influence on the student attitudes. Were the students taught from the beginning to think in terms of code reuse and with an obligation to fully understand the requirements in the form of tests? The lecturer records that test-driven development was taught but wondered whether the students understand how to translate requirements to tests and what was involved in that process? In all of these cases, the students were thinking about what was expected of them and the criteria were influencing the judgement of the students.

The lecturer in recording the instances tended to focus on the difficulties and recorded less of the positives. This was so that the lecturer could endeavour to address these issues in the next offerings of the papers. The positives that were recorded reflected comments of appreciation for the lecturer's teaching style or what had been learnt. These notes showed that the students recognised that this lecturer's papers were difficult but that the students felt that they learnt a significant amount.

7 Discussion

Like any marking criteria, there needs to be alignment with the specification of the task. In the SOLO taxonomy marking strategy and the cases, there is a consistent strategy of rewarding valued programming practices and placing of an emphasis on the quality of the solution. The alternative may be to emphasise achieving a solution with quality as an option. A further study on this is to be completed as part of research currently being carried out on improving learning in programming.

The SOLO taxonomy based marking criteria have been used with programming assignments that involved the implementation of a reasonable sized program. Can it be applied to smaller projects where there may not be so much duplication or shared functionality in the code? The research work suggests that we can use it to evaluate student responses to reading and comparing code segments (Whalley et al., 2006, Thompson et al., 2006) and in evaluating the strategies used to solve code reading problems (Lister et al., 2006). Work is continuing on establishing the use of the SOLO taxonomy to assess the writing of code segments for exam conditions.

What is clear is that there is a need to ensure the project includes issues that enable solutions that are dependant on more than a single or limited range of concepts. Larger projects should provide for the possibility of implementing a solution as independent pieces or as an integrated whole. Biggs (1999) describes how the SOLO taxonomy can be used to write assessment items.

8 References

Biggs, J. B. (1999) *Teaching for quality learning at University,* Buckingham, Open University Press.

Biggs, J. B. & Collis, K. F. (1982) *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome),* New York, Academic Press.

Dijkstra, E. W. (2000) Answers to questions from students of Software Engineering. From http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1305.PDF.

Felleisen, M., Flatt, M., Findler, R. B., Gray, K. E., Krishnamurthi, S. & Proulx, V. K. (2001) *How to design class hierarchies: Object-oriented programming and computing*.

Hattie, J. & Purdie, N. (1998) The Solo model: Addressing fundamental measurement issues. In Dart, B. & Boulton-Lewis, G. M. (Eds.) *Teaching and learning in higher education.* Camberwell, Vic, Australian Council of Educational Research.

Lister, R., Simon, B., Thompson, E., Whalley, J. & Prasad, C. (2006) Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *Innovation and Technology in Computer Science Education (ITiCSE 2006)*. Bolonga, Italy.

Maki, P. (2004) *Assessing for learning: Building a sustainable commitment across the institution,* Sterling, VZ, Stylis Publishing.

McBreen, P. (2001) *Software craftsmanship: The new imperative,* Boston, Addison Wesley.

McConnell, S. (2004) *Professional software development: shorter schedules, higher quality products, more successful projects, enhanced careers,* Boston, Addison Wesley.

November, P. (1996) Journals for the journey into deep learning: a framework. *Higher education research and development,* 15**,** 115-127.

November, P. (1997) Learning to teach experientially: a pilgrim's progress. *Studies in Higher Education,* 22**,** 289-299.

Proulx, V. K. & Gray, K. E. (2006) Design of class hierarchies: An introduction to OO program design. *Inroads - The SIGCSE Bulletin,* 38**,** 288-292.

Thompson, E. (1997) *71253 Systems Design and Development,* Lower Hutt, The Open Polytechnic of New Zealand.

Thompson, E. (1998) Delivering education that satisfies industry. *NACCQ National Conference.* Auckland, National Advisory Committee on Computing Qualifications.

Thompson, E. (2004) Does the sum of the parts equal the whole? In Mann, S. & Clear, T. (Eds.) *Proceedings of the seventeenth annual conference of the National Advisory Committee on Computing Qualifications.* Christchurch, New Zealand, National Advisory Committee on Computing Qualifications.

Thompson, E., Whalley, J., Lister, R. & Simon, B. (2006) Code Classification as a Learning and Assessment Exercise for Novice Programmers. In Mann, S. & Bridgeman, N. (Eds.) *The 19th Annual Conference of the National Advisory Committee on Computing Qualifications: Preparing for the Future — Capitalising on IT.* Wellington, National Advisory Committee on Computing Qualifications.

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, A. & Prasard, C. (2006) An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In Tolhurst, D. & Mann, S. (Eds.) *Eighth Australasian Computing Education Conference (ACE2006).* Hobart, Tasmania, Australia, Australian Computer Society Inc.