Ritter, N., McGill, T.J. and Thompson, N. (2009) Incremental submission of programming code using object-oriented classes. ACM SIGCSE Bulletin, 41 (4). pp. 66-70.

Incremental Submission of Programming Code Using Object-Oriented Classes

Nicola Ritter, Tanya McGill, and **Nik Thompson**
School of Information Technology
Murdoch University
Murdoch WA 6150, Australia nritter@globaldial.com, [t.mcgill, n.thompson]@murdoch.edu.au

Abstract: Given increasing adoption of agile software development methodologies it is essential that information technology students are exposed to them. This paper describes and evaluates an attempt to introduce agile programming into a core second year programming course. The initiative appeared to be associated with improvements in both drop out and pass rate, and student perceptions of the innovation were largely positive.

Categories and Subject Descriptors: K3.2 [Computers and Education]: Computer and Information Science Education-*Computer Science Education;Curriculum*
General Terms: Design, Measurement
Keywords: Agile methods, software development, software engineering education

1. INTRODUCTION

There has been a shift in work practices within the software engineering industry in the recent years. A new generation of software development methodologies, known as agile methodologies, have gained in popularity. Examples include Extreme Programming (XP) and Scrum. These approaches to development place a strong emphasis on iteration; design, coding and testing are done repetitively with re-work as necessary. There is also a much greater emphasis on on-going testing than in previous methodologies. Team work is encouraged, and there is a focus on rapid delivery of quality software. Agile methodologies are considered light-weight and impose less process burden upon the developers and because of their capacity to deal with volatile requirements they have received acclaim from practitioners [Reifer 2002].

Given the increased interest in, and adoption of, agile software development methodologies it is essential that information technology (IT) students are exposed to them. The IEEE/ACM Computer Science - Software Engineering Curriculum [2004] lists agile concepts and several agile practices (such as refactoring, test-driven development) as essential topics, but they have yet to be fully embraced by academic institutions. The study described in this paper, addressed this issue by evaluating an attempt to introduce agile programming into a core second year university programming course.

2. LITERATURE REVIEW

A number of authors have discussed the issues associated with the teaching of agile development processes. Hazzan and Dubinsky [2007] argue strongly for increased teaching of agile software development. Their reasons include the fact that the approach evolved in and is used in industry and that it emphasises teamwork and the human aspects of software development. They also argue that it supports learning processes, encourages diversity, emphasises management skills and provides a single teachable framework. Schneider and Johnston [2005] argue however that while exposure to agile practices is beneficial when learning about small-scale development, they do not believe that agile approaches are appropriate for learning about large-scale system development.

The descriptions in the literature of experiences introducing agile software development in tertiary courses appear to be largely positive. Melnik and Maurer [2005] investigated the perceptions of a wide range of students who had been exposed to agile programming, and found that students were very positive about core agile practices and keen to continue to use agile methods in the workplace after graduation. Kessler and Dykman [2007] combined a traditional software engineering focus and an agile process in the domain of PDA programming using C#. The agile process was the last phase of the course and was worth 30%, and designed to be very hands-on. They concluded from the positive student feedback that they had found a good balance. Similarly, Layman, Cornwell and Williams [2006], Loftus and Ratcliffe [2005] and Sherrell and Robertson [2006] all reported positive student perceptions and evaluations of classes when they introduced agile methods.

3. BACKGROUND TO THE PROJECT

The Data Structures and Abstractions course is considered to be one of the hardest IT courses at the university. It has a high failure and withdrawal rate independent of the quality of the teacher (as measured by student evaluations). It covers C++; object orientation and class design; data structures from stacks to B+ trees; and algorithms from linear search to quicksort, including discussion of algorithmic techniques such as back-tracking.

In a traditional manner, this course has been assessed with one or two small assignments plus a major assignment and an exam. This assessment has caused many problems. Students do three courses in a semester. Due to the need to teach students sufficient material before they do a major assignment, students inevitably end up with three major assignments due in the last 1-2 weeks of semester. This means that the students have an almost intolerable burden, leading to high levels of stress and high rates of late semester drop-out.

A second problem is the quality of the work produced by the students. The students need to code large numbers of classes to do anything significant for the major assignment and end up losing marks because they make the same mistakes in each class. To 'save time' they often have poor class split-up as they think that the more classes they have the longer it will take to code. They do little unit (class) testing and what they do is only because marks have been awarded for a test plan. They generally do design *after* they have finished the code; not as a tool for getting things right, but again as a way of fulfilling requirements. In other words the assignment is truly an assessment tool and not a learning tool; this is frustrating for both student and teacher alike.

The aim of this study was to see if we could successfully apply agile development to the assessment process in this course, and hence solve the assessment problems described above, as well as to introduce students to agile software development.

4. DESCRIPTION OF THE STUDY

4.1 Changes to the Course

The previous type of assessment was replaced with a single project that involved building the business (back-end) part of a DVD Collection program. The students were required to work with C++ and were not allowed to use a database or any of the STL classes other than string. As students rarely write pseudo-code in advance it was decided that design would consist of UML class diagrams, test plans, version information within class header files and pseudo-code as comments within the program where they had a need to do design before coding.

In the first week students were taught about UML class diagrams and incremental programming techniques and given the coding standards to be followed for the course. The UML class design covered how to decide on classes, rules of inheritance and composition etc., and UML class diagram notation. The incremental programming covered the building of an individual class. Each class was considered to need a test plan, header file, implementation file and a test program. Each method within the class was then to be coded by deciding on what tests would confirm it worked and recording them in the test plan, coding the method, and then implementing code within the test program that would perform the tests listed in the test plan. This was to be done before coding the next method.

In the first and second weeks of the twelve week course the students then practiced producing a UML class design for specific problems. Their initial class diagram for their DVD collection program was then due at the end of the second week. This first class diagram was worth 10% of the marks for the project and was returned within a week to the students with detailed comments about where it was wrong and where it could be improved.

Thereafter students were required to submit, at their own pace, each class as it was completed. Each submission was required to include an updated and corrected class diagram, a spreadsheet containing all the test plans for the project—each class in its own named worksheet—plus the three code files: header, implementation and test.

Each submission was then marked within two days and returned to the student. If the class was very poorly done, students were required to resubmit it before it was marked. If it had medium level problems they were required to resubmit a corrected version before submitting more classes. If it had minor problems then they were allowed to resubmit it with the next class. All resubmitted or new work that did not correct previous mistakes was rejected out of hand.

Each updated class diagram was worth 2 marks and each class was given a mark based on its adherence to the course standards, coding efficiency and the quality of its test plan. Correctness of the code was judged by the comprehensiveness of the test plan and checking that the test program's output exactly matched the expected output in the test plan. The mark for the class was then weighted based on its complexity, ensuring that a simple class such as "Person" was worth less marks than a container class. Similarly a container that used a more complicated data structure was worth more marks than one that did not. Finally, a class that was similar to either a previously submitted class or one provided in the lectures was given less marks. Marking was severe: students were required to get it right, not allowed to be nearly good. Students were allowed to keep submitting classes until they scored 100% in total for the project.

4.2 Evaluation of the Changes

Ninety three students were initially enrolled in the course. The changes to assessment applied to all students. Evaluation of the changes was undertaken in several ways. Changes in drop out rate and pass rate were considered. Students' perceptions were also obtained via a web survey. Several weeks before the semester ended all students were emailed inviting them to participate in the evaluation of the

introduction of agile programming by *completing a questionnaire on the web.* Completion of the questionnaire was voluntary and all responses were anonymous. The questionnaire included the following types of questions.

- A series of statements listing the assumed positive aspects of the changed assessment were presented and students were asked to rate their agreement with them on a Likert scale labelled from 1 (Strongly Disagree) to 5 (Strongly Agree).
- A series of statements about the way the assessment process was handled were also presented and students were asked to rate their agreement with them on a Likert scale labelled from 1 (Strongly Disagree) to 5 (Strongly Agree).
- Students were also encouraged to provide additional comments about the assessment and their experiences with it.

5. FINDINGS

5.1 Retention and Pass Rate

The percentage of students who dropped out was only 17% in contrast to the previous year where the drop out rate was 22%. The percentage of students who passed the course went from 46% in the previous year to 60%, *with no lowering of standards*. This was due to the fact that students got *much* higher marks in the assignment than in prior years: the agile methodology was very successful in teaching students to build a large complicated system. In particular, middle-of-the range students, who used to struggle with an assignment of this scope, not only passed but excelled when using the agile paradigm.

5.2 Instructor Perceptions

From a course coordinator's perspective the results of using the agile paradigm were mixed. To make it work there has to be a guarantee that students will get back a checked class quickly: the aim was to return work within 2 days. This requirement meant that tutors had to do an average of 1 hour marking *every day*. This does not sound that much, and indeed equates to the total amount of time spent on marking in previous years. However the relentless nature of it became tiring towards the end of semester. Furthermore it affected research output as it made it difficult to concentrate on research for a whole day: students complained bitterly if the 2 day deadline was not met. Part-time tutors found it nearly impossible to adhere to, and yet the fast checking of work quickly is central to the concept.

A second problem was the difficulty in explaining the methodology to the part-time tutors. Even after insisting that they use the agile assessment whether they liked it or not, it was found that at least one tutor regularly broke the marking rules and marked students down for resubmitting work, even though that was what they were required to do! This meant that the course coordinator had to do regular moderation at a greater depth than is normal in such a course.

Despite these problems the rewards of using agile methodology were great. For the first time students were learning rather than simply being assessed. There was great satisfaction in insisting that students get it right rather than 'near enough is good enough'. It was much easier to help the struggling middle-of-the-range students than in the past because they were taking small steps, not trying to 'leap to the top of a building in a single bound'. The top students excelled. They leapt into the project well in advance of everyone else and completed it—doing advance reading and study—by the end of the first half of the semester. Furthermore this meant that problems encountered in the project description itself were highlighted, and hence clarified, long before the average student came across the problem.

Since students were required to submit parts of the project regularly they, and the teaching staff, were able to track progress much more easily than in the past. Students who were not submitting could be talked through the problems they were encountering and encouraged to get a single class in to get themselves going.

The final advantage of the system from a course coordinators point of view was the reduction in cheating. Students did their own initial designs and no attempt to enforce uniformity was made. This meant that every student had a slightly different model. Some wanted to include detailed information about actors, some wanted to include documentaries and TV-Series as well as just 'DVD', and no two people had exactly the same set of attributes for every class. If two students had had the same model it would have been very obvious, allowing collusion to be picked up early. From then on students were essentially working on individual projects which made cheating nearly impossible.

5.3 Student Perceptions

This section describes the students and the feedback received from them. Thirty six students completed the online questionnaire. Of these students, 13 (36%) had worked in the IT industry as programmers, but only 3 (8.3%) had written C++ as part of this employment.

Table 1 provides summary information about the aspects of the assessment that students valued. In general students were very positive about the changes. The comment below encapsulates the positive nature of the feedback

> *I definitely got more out of this project than any other assessment in the degree so far, and it has made this course the most interesting I have taken.*

Table 1. Positive aspects

	Mean	Min.	Max.	Std. Dev.
The ongoing assessment in this unit has made it easier to fit this unit around other units.	4.53	1	5	0.91
Software is easier to design if it is coded incrementally as done in the assessment project.	4.39	1	5	0.99
The ongoing assessment in this unit has made it easier to fit this unit around other commitments such as work.	4.36	1	5	1.07
Being able to resubmit poor work for extra marks was very useful.	4.33	3	5	0.76
An ongoing project was useful experience for you for when you work in industry.	3.94	1	5	1.09
I learnt more from this type of assessment than from the normal type of assignment.	3.75	1	5	1.20
Separate assignments are easier to do than one ongoing project.	2.50	1	5	1.207

In addition a number of students specifically commented about the applicability of the nature of the assessment to industry.

> *This was easily the best project I've ever received. I'm a programmer by trade, I work on webapps in PHP. All of my work is 'agile' in nature, I find that the waterfall method is out of date, time consuming and just causes massive issues at the end when you have to test everything at once, rather than testing as you go.*

> *…the more the practical work reflects industry methods better.*

In particular students liked the flexibility that the ongoing assessment provided, and they found it easier to design software in an incremental fashion.

> *I thought the ongoing assessment was a great idea as it made the project seem easier as it was broken up into manageable problems rather than one large problem. It also gave the opportunity to get constant feedback in relation to how you were going and gave you a sense of achievement as you saw your mark increase.*

Students also noted the value of iterative development.

> *In previouscourses where there is only one big assignment at the end I found I never actually learned how to correct the mistakes I made, just moved on. However this form of assessment forces you to get code not only "correct" but clean also.*

Table 2 includes more specific feedback about the way the assessment process was handled. All students were very conscious of the need for rapid feedback, and there was wide range of responses to the question relating to the degree to which they believed this had been achieved. This is attributable to the fact that some of the tutors were part-time and had difficulty meeting the specified turnaround times.

> *….projects were sometimes slow to be marked making it difficult to correct the projects and resend them in a quick enough manner...*

Table 2. Feedback on implementation

	Mean	Min.	Max.	Std. Dev.
Fast feedback is crucial to the usefulness of this type of assessment.	4.67	3	5	0.59
The tutors gave feedback quickly enough to be useful.	3.64	1	5	1.25
The process of resubmission of the assignment worked well.	3.97	1	5	1.11
Having to get things completely right before moving on was annoying.	2.83	1	5	1.36
Having to get the structure of the directories and code correct was annoying.	2.58	1	5	1.20

Students were also specifically asked about whether the need to get code and directory structures correct before moving on was an issue. A wide range of responses was provided, but for the majority of students this did not seem to be a big issue.

However students did recognise the danger of not having fixed submission dates:

> *The negative side that I see to this unit is that often, especially for me, it gets neglected for other units. I prioritize my time for each unit and with the open submission I often push work back for the unit in place of other units work/assignments were they do have a deadline. So being able to set weekly or fortnightly goals either individually or set by the course outline to submit something would greatly help in the unit. Unfortunately you then lose the open submission status. So in conclusion students attempting this will need to be able to maintain their own learning.*

6. CONCLUSION

This research evaluated an attempt to introduce agile programming into a core second year programming course. It considered the impact on student drop out and pass rate, and explored student perceptions of the use of agile methods in their major project. The changed type of

assessment appeared to be associated with improvements in both drop out and pass rate, and student perceptions of the innovation were largely positive. The major problem identified was the difficulty returning marked work within the shorter timeframe.

REFERENCES

[1] Hazzan, O. and Dubinsky, Y. 2007. Why software engineering programs should teach agile software development. SIGSOFT Software Engineering Notes, 32 (2). 1-3.

[2] IEEE/ACM. 2004. Software Engineering 2004: Curriculum guidelines for undergraduate degree programs in software engineering. Available from http://sites.computer.org/ccse/SE2004Volume.pdf

[3] Kessler, R. and Dykman, N. 2007. Integrating traditional and agile processes in the classroom. Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, 312-316.

[4] Layman, L., Cornwell, T. and Williams, L. 2006. Personality types, learning styles, and an agile approach to software engineering education. SIGCSE Bulletin, 38 (1). 428-432.

[5] Loftus, C. and Ratcliffe, M. 2005. Extreme programming promotes extreme learning? Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, 311-315.

[6] Melnik, G. and Maurer, F. 2005. A cross-program investigation of students' perceptions of agile methods. Proceedings of the 27th International Conference on Software Engineering, 481-488.

[7] Reifer, D.J. 2002. How good are agile methods? IEEE Software, 19 (4). 16-18.

[8] Schneider, J-G. and Johnston, L. 2005. eXtreme Programming: helpful or harmful in educating undergraduates? Journal of Systems and Software, 74 (2). 121-132.

[9] Sherrell, L.B. and Robertson, J.J. 2006. Pair programming and agile software development: experiences in a college setting. Journal of Computing in Small Colleges, 22 (2). 145-153.