Routledge
Taylor & Francis Group

Innovation in Teaching and Learning in Information and Computer Sciences

ISSN: (Print) 1473-7507 (Online) Journal homepage: http://www.tandfonline.com/loi/rhep14

Can Individual code reviews improve solo programming on an introductory course?

Dr. Glenn L. Jenkins & Dr. Oluwakemi Ademoye

To cite this article: Dr. Glenn L. Jenkins & Dr. Oluwakemi Ademoye (2012) Can Individual code reviews improve solo programming on an introductory course?, Innovation in Teaching and Learning in Information and Computer Sciences, 11:1, 71-79

To link to this article: http://dx.doi.org/10.11120/ital.2012.11010071

CAN INDIVIDUAL CODE REVIEWS IMPROVE SOLO PROGRAMMING ON AN INTRODUCTORY COURSE?

Dr. Glenn L. Jenkins
Swansea Metropolitan University
Swansea
SA1 6ED
glenn.l.jenkins@smu.ac.uk
http://www.smu.ac.uk/research/index.php/dr-glen-l-jenkins

Dr. Oluwakemi Ademoye
Swansea Metropolitan University
Swansea
SA1 6ED
kemi.ademoye@smu.ac.uk

ABSTRACT

Peer code reviews have been successfully applied to the teaching of programming and can be applied to solo programming. Collaborative approaches are currently popular and have been successfully applied though social interaction and assessment issues limit their application. It is believed that a checklist based individual code review can provide a framework which allows students to proofread their code prior to submission, improving performance. Pilot and follow-up studies were conducted at Swansea Metropolitan University and although the results are inconclusive some important observations are made with regards to the use of this technique. Further study into the effects of individual code reviews on student performance is recommended.

Keywords

Programming, Pair Programming, Code Review, Personal Software Process.

1. INTRODUCTION

Programming is a difficult subject for many students and therefore a popular topic in computing education research, with extensive research into the learning and teaching of programming (Sheard et al, 2009). Collaborative learning approaches are desirable as they fit well with the needs of industry (Hundhausen et al, 2009). Such approaches have recently been used (Huggins, 2009, Hundhausen et al, 2009, Sheard et al, 2009) to combat high attrition rates common on programming courses (Hundhausen et al, 2009, Trytten, 2005) along with pair programming (McDowell et al, 2002, McDowell et al, 2006, Williams and Upchurch, 2001) and other techniques (Doyle, 2005).

Recently peer code reviews also known as code inspections have been successfully applied to the teaching of programming (Hundhausen et al, 2009, Trytten, 2005, Wang et al, 2008). Code reviews can also be applied in an individual context as in the Personal Software Process (PSP) (Humphrey, 1997). Students at present undertake pair programming which has an element of code review (Hundhausen et al, 2009) and work on a number of collaborative exercises on this basis. The utilisation of pair programming raises a number of concerns, firstly pair breakups and their effect on retention (Jacobson and Schaefer, 2008). Secondly difficulties associated with identifying the contribution of individual students to joint assignments (McDowell et al, 2006) and finally the effect on progression. Some students may struggle to make the transition between a first year of pair programming and solo programming on later modules (Simon and Hanks, 2008). The compromise is a mixture of independent and paired assessment as suggested by Jacobson and Schaefer (2008) encouraging the development of both pair and individual programming skills.

Research suggests the review of code can be used to improve students' grasp of programming. The question becomes can this process be employed to improve solo programming? In the following section we will review the literature on software inspections, related techniques and their use in pedagogy. This will be followed by the experiment methodology, recorded results and finally a discussion of these results and conclusion.

2. BACKGROUND

All software progresses through a development process which normally includes some kind of analysis (identifying needs), design (specifying software to meet the needs identified), implementation (the creation of the software according to the specification) and finally testing (ensuring the software developed meets those

needs) (Sommerville, 2007a). Verification and validation are integral parts of the software engineering process and may be applied at various stages (Boehm, 1979, Sommerville, 2007b) including implementation, the focus of introductory programming modules. Boehm (1979) succinctly expresses the differences between these in terms of two questions: validation asks, “Are we building the right product?”, whereas verification asks, “Are we building the product right?”

Verification and validation in the implementation stage are implemented by two complementary processes; software inspections (or peer reviews) and software testing (Sommerville, 2007b). Software testing involves running an implementation of the software or a component part, with test data and comparing the output or behaviour of the system to ensure it matches what is expected. It is a dynamic technique for verification and validation (Sommerville, 2007b). The introduction of testing in computing courses is an area of research in its own right however it has been observed that students focus on the output i.e. write code to meet the requirements of a limited number of tests (Edwards, 2003b), to the detriment of other considerations such as design, commenting, appropriate use of abstraction etc. (Edwards, 2003a).

Software inspections are, “…reviews whose objective is program defect detection” (Sommerville, 2007b). They provide a static test of the code (i.e. the code is not running as with testing which is a dynamic process). This has advantages over testing in that errors often mask other errors (Sommerville, 2007b). For example Function A calls Function B passing a single value. During testing it is found that Function B operates incorrectly when it is passed a value outside the expected range. The problem is solved by ensuring Function B ignores out of range values passed from function A, however this does not fix the error in the Function A. Individual inspections usually uncover more errors than testing but as they require lengthy meetings between a number of developers they are also more expensive in terms of time and resources (Sommerville, 2007b). Sommerville (2007b) highlights studies which suggest that 60% more defects can be found than with testing and that programs can be made 90% defect free. In light of this a number of formal inspection methods have been developed.

The notion of formal inspection originates from IBM in the 70’s and is attributed to Fagan (1976) whose original method and its variations have been widely adopted (Sommerville, 2007b). In general software inspections are a formal process carried out by four or more individuals, though variations include more roles which may be shared by a single person (Sommerville, 2007b). Fagan’s original proposal suggests the roles of ‘Reader’ (who reads the code aloud), ‘Tester’ (who reviews the code from a testing perspective), ’Author’ (the person who wrote the code and will make any changes) and the ‘Moderator’ (who manages and organises the process) (Fagan, 1976). All members of the team perform the review considering the code on a line by line basis while the author notes the changes to be made (Sommerville, 2007b).

There is no accepted definition of the software inspection process and terminology used varies between sources (Aybuke et al, 2002). Humphrey (1997) considered code reviews as individual and peer code reviews as software inspections, while Turner et al (2008) indicates that code reviews are a kind of peer review where code is the subject. Inspections have been distinguished from reviews and walkthroughs by the IEEE, whose standards indicate that an inspection’s objective is to identify defects and deviations from standards (Aybuke et al, 2002). While the objective of technical reviews and walkthroughs is to evaluate the software product. The process as conducted by organisations or individuals though described as a software inspection may span the IEEE definitions (Aybuke et al, 2002).

Checklists of common programming errors may be used during an inspection to focus the discussion, based on examples from books or checklist examples (Sommerville, 2007b). Checklists are specific to the language being used and reflect the standards and practices of the organisation (Sommerville, 2007b). Checklists are a non-systematic reading technique used to help identify errors during the review process, an improvement over ad-hoc reading (Aybuke et al, 2002). They are intuitive unlike their systematic counterparts (e.g. stepwise abstraction, scenario-based reading, defect-based reading and perspective reading) (Aybuke et al, 2002). In empirical studies comparing the various reading techniques checklists have been shown to be as effective as other methods, while being intuitive, requiring minimal training (Aybuke et al, 2002) and importantly effective in the hands of inexperienced individuals (Muller, 2005).

McDowell et al (2002) describe pair programming as two programmers of varying experience sitting side by side at a single computer developing a program together. One programmer identified as the “driver” will control the input devices (mouse and keyboard) and actively creates the code, the “non-driver” or “navigator” (Bagel and Nachiappan, 2008) checks the code produced identifying misspellings, logical and syntactical errors and ensuring that the code is in keeping with the design (McDowell et al, 2002). After a designated

period the programmers exchange roles, possibly moving to work with others within the programming team (McDowell et al, 2002).

Due to concerns over pair breakups and their effect on retention and assessment (Jacobson and Schaefer, 2008) and concerns over the contribution of individual students to joint assignments (McDowell et al, 2006), it is suggested that peer assessment (Williams, 2007) and more technological solutions such as managed code repositories with code check-in and e-portfolios are used to ensure equal contribution (Chapparro et al, 2005). It has also been suggested that some students may struggle to make the transition between a first year of pair programming and solo programming on later modules (Simon and Hanks, 2008). The compromise is a mixture of independent and paired exercise and assessment, as suggested by Jacobson and Schaefer (2008) encouraging the development of both pair and solo skills.

Research suggests that pair programming in industry can reduce or even remove the need for code reviews (Phongpaibul and Boehm, 2006) as every line of code is reviewed as it is written by the "non-driver" (Williams and Kessler, 2000). Advocates of reviews as a pedagogical tool suggest that this is a kind of peer review providing justification for its further exploration (Hundhausen et al, 2009), two person reviews, where one person is the author, have also been shown to be effective (Aybuke et al, 2002). The effectiveness of reviews and their close cousin the software inspection has been demonstrated by a number of researchers.

Peer reviews are used in a number of educational contexts partly as they are an authentic real-world practice in many veins of computing and as such prepare students to accept and deliver constructive criticism as well as *soft* skills such as negotiation and team work (Hundhausen et al, 2009). They also promote opportunities for deep learning that is learning and assessment at the highest level of Bloom's taxonomy: *evaluation* (Hundhausen et al, 2009). Students may also gain from this process in appreciating that there are multiple correct solutions and come to see their peers and themselves as legitimate sources of knowledge (Hundhausen et al, 2009). Anecdotal evidence from a number of studies suggest that working together promotes the discussion of programming issues (Hundhausen et al, 2009) and encourages students to consider other solutions (Turner et al, 2008).

Trytten (2005) employed a peer code review and observed a number of improvements. Trytten's approach evolved from students reviewing their peers' code to the review of sanitised code submitted to and augmented by the tutor as required. The review process consisted of teams of students with a list of multiple choice (true or false) questions which related to the code they were to review, ranging from low level questions about syntax and naming conventions to high level design questions as the course progressed. Anecdotal evidence from the study suggests students improved in regards to their attitude to programming and communication skills. As the students are attempting to indentify defects this may be considered an inspection despite the terminology used.

Wang et al's (2008) research focuses on a loose implementation of formal inspections where individual student participate in set groups as either author or reviewer and communicate via email (sending both code and a review form). These emails are copied to the tutor so that the tutor receives the original and revised code along with the review form. The focus of Wang et al's research is on improving the process. Improvement in student learning and code quality are assumed based on the literature and previous research.

Wang et al's observations highlight a number of problems. Student based problems included communication (missing forms and failing to send copies to the tutor), attitudes (missed deadlines and partial participation), mismatch of students (good students frustrated by reviewing very broken code and poor students struggling to make meaningful comments on very good code) and conspiracy (independently reviewing code to minimise the errors found). Problems for the tutor included an increased workload (reviewing the emails) and difficulties with the control of the process (ensuring code and forms were exchanged appropriately and at the appropriate time). Among the suggested solutions to these problems are the ranking of students prior to grouping to ensure that groups contain peers and re-assigning groups or sanitizing submissions to prevent conspiracy both of which lend themselves to online tools.

Hundhausen et al (2009) produced strong preliminary results using pedagogical code reviews. Their technique was based heavily on those used in industry though the process was simplified due to time constraints. Students were placed into review groups using previous grades to give a spread of abilities and allocated a graduate student as coordinator. Each graduate student was paid a nominal sum for their assistance. The groups met and reviewed sections of code observing the roles used in industrial code reviews (author, reader, inspector and recorder). At the end of each section the roles were cycled and a

section selected from another student's code. Their findings suggest that students found fewer defects between the first of the three reviews and the last, also the kind of defects changed from simple syntax and style issues to more implementation and design based concerns. This was reflected both in an analysis of the logs of the meetings and the exit questionnaires after each review.

Turner et al (2008) used the code review process with students as individuals rather than in groups. The students reviewed a 'good' and 'bad' example of a solution to an object orientated programming problem. The students were provided with a simple *rubic* via the Moodle Virtual Learning Environment and reviewed the two examples. The study found that despite previously undertaking object oriented programming there were a number of misconceptions amongst the students identified by an analysis of outliers in the student responses.

Earlier research by Humphrey (1997) introduced code reviews as part of the Personal Software Process, or PSP, a development mechanism developed at Carnegie Mellon University with the aim of producing undergraduates with disciplined personal software development practices. The process suggested involves reviewing a printed copy of the code against a short checklist of possible errors prior to compilation (Humphrey, 1997).

The process of reviewing code has educational benefits requiring students to operate at the highest levels of Blooms taxonomy, specifically students must analyse and evaluate the code they are reviewing (Turner et al, 2008). In terms of current theories in adult learning such as Kolb's learning cycle (Petty, 2004b) or Race's Ripples model (Race, 2005) reviewing the code provides students with an opportunity to gain feedback by checking their code against a checklist. Brunner describes "Scaffolding" as a process whereby a mental stepping stone allows the student to enter their Zone of Proximal Development (Brunner, 1985). This is the region of learning between what the student can at present achieve alone and that which they can achieve with help (Panselinas, 2009). During the review process students may consider alternatives effectively using their existing solutions as a starting point for the development of alternatives, effectively self-scaffolding.

The integration of pair programming into the first year programming module at Swansea Metropolitan University (SMU) has produced many of the benefits identified in the literature. Pair programming has been shown to increase retention (McDowell et al, 2002), confidence (Williams and Upchurch, 2001), enjoyment (McDowell et al, 2006) and improve program quality. While observed benefits include; a broader range of solutions (Williams and Upchurch, 2001), and improved group communications (Williams and Upchurch, 2001). Support and encouragement is provided by the partner and a sense of duty or camaraderie (Ma et al, 2005, Williams and Upchurch, 2001).

In pair programming based assessment it is difficult to accurately assign marks to individual students (McDowell et al, 2006) a common problem with any group based assignment (Reece and Walker, 2007). There are also concerns that students may struggle with the transition to solo programming in future modules (Simon and Hanks, 2008). As with any group work there may be issues either with the activity, for example students taking little interest in or responsibility for the work, or social issues for example off topic discourse, arguments and violence (Armitage et al, 2007, Petty, 2004a). Jacobson and Schaefer (2008) suggest using a mixture of individual and group assignment as a compromise.

Code reviews and peer code reviews have been applied in educational contexts with positive results, improving both low level skills such as style and syntax and high level skills such as design and defensive programming (Hundhausen et al, 2009). Formal code reviews increase workload of the tutor who has to arrange, manage and monitor the groups (Turner et al, 2008). Personality clashes are a major issue with the peer review process (Trytten, 2005) especially in student inspection groups where all members are peers (Phongpaibul and Boehm, 2006). There are also issues with student contribution, students failing to engage causes a number of issues largely dependent on their role within the review (Wang et al, 2008). Students may also conspire to, "blunder", through the process (Wang et al, 2008).

Making the review process individual can eliminate the problems associated with group and pair work as the student is working alone. Research has been conducted into individual code reviews and their improvement on solo programming. Humphrey (1997) provides some evidence that students using PSP which includes a code review show improved development skills. This suggests individual reviews may be well suited to solo programming assignments. However PSP itself is a formal, documentation heavy technique, focused on recording various metrics (Bloch, 2000). These metrics include the time spent writing the program, the number of lines of code, the number of defects etc. are then used to gauge its size and complexity as a basis

for future estimation (Bloch, 2000). This lengthy documentation is a cause of resistance amongst some students and instruction in the process is required (Bloch, 2000).

The aim of this research is to ascertain whether individual code reviews based on checklists like those used in PSP (Humphrey, 1997) and during formal code inspections in industry (Sommerville, 2007b) with minimal reporting can be used to improve solo programming. This will allow students to systematically check their work as they would have done for pair programming and complement the pair programming approach currently used within the module.

3. METHODOLOGY

The vehicle for this experiment was the first year programming module for BSc Software Engineering and BSc Games Development students entitled Introduction to Programming. This is a key module for both courses and is taught initially in C with students moving to C++ when objects are introduced. An initial pilot study was conducted as follows: the students were split into two groups (17 students in each) a test group and control group, with alternating students being picked from a sorted list based on the first assignment marks. The test group were provided with a checklist based on widely available code reviews (Brykczynski, 1999) and teaching material along with some brief training. The students were also tasked with recording which of the points help them to identify defects in their code (through simple annotation of the list rather than a formal reporting process). Unlike the test group the control group were not provided with the checklist. The pilot study was developed as part of a Postgraduate Certificate in Education (PGCE) module and limited due to the size of the sample. The study did reveal several important points notably that the training provided had been insufficient and the students needed practice using the code reviews to use them with confidence.

Following on from the pilot study and over a period of two consecutive years, two cohorts of students on the first year programming module were instructed in the use of code review from the first lesson. Each topic added more points to the checklist which students used in tutorials and in their first assignment. Students were taught using this approach, gaining experience in using the checklist to review their code through tutorials and the first assignment. A second assignment was also given as part of the assessment for the module. The follow up study focuses on the second assignment given. The 35 students in the first cohort group, from the first year of the study, made up the control study group and were not asked to review their code, while the 38 students who took the module the following year made up the checklist group and were asked to review their code. For the checklist group, five marks were associated with the submission of a table indicating the points of the code review the students had used.

A number of metrics for assessing student performance are suggested by the literature. Brykczynski (1999) and others use student grades to compare performance while Hundhausen et al (2009) studied the kind of problems uncovered by the review over a series of assessments accompanied by a survey to gauge the opinion of students. The average mark of the group on the current assessment is used to quantify the results, accompanied by a survey which provides an insight into the student's perception of the process for future refinement.

4. RESULTS

For the pilot study a comparison of the two groups' performance shown in Figure 1 indicates that the test group increased their performance by a small margin as a result of undertaking the code review. A statistical analysis of these results however indicates that they are not statistically significant ($t=0.40$, $p > 0.05$). It is also noted that a large proportion of the cohort failed to submit, 13 of the 17 students in the control group submitted while only 8 of those in the checklist group submitted. The survey results also indicate that at least two of the students in the checklist group did not use the code review.

	Average	**Std. Dev.**	**% Non Submission**
Control Group	56.35	13.88	23.53
Checklist Group	59	15.84	52.94

Figure 1. Pilot study: Checklist group vs. Control group

The survey conducted after the submission of the assignment consisted of 10 questions following a similar format to that laid out by Chapparro et al (2005) in their work on pair programming. The group asked to perform the code review undertook the survey and 6 of the 8 who submitted responded. The survey results showed that the majority of students in the test group undertook the code review and those students generally

agreed that it improved the quality of the programs they produced and that they would consider using them for future assignments. The survey suggests that the majority of students found the most positive element of the code review process was the improvement of their submissions. It also suggests that the code review helped in highlighting points that students had overlooked or were not aware of.

The survey result suggest that some students struggled to apply the code review; these included problems with the concepts, terminology and the process especially with regards to when it should be applied and how it should be reported. It also shows that the majority of students that did not undertake the code review but had the opportunity to do so overlooked the code review while concentrating on the assignment itself. It was observed that a number of students asked questions about items on the checklist and interestingly a number saw the review as a form of structured proofreading for programs.

The follow up experiment took place in the two years following the original pilot. Comparing the results for assessment one, for which the delivery of material and assessment were kept the same as far as possible there is little difference in the averages i.e. no statistically significant difference between them ($t = 0.26$, $p > 0.05$). For the second assignment one group were asked to use the code review checklist with a handful of marks for doing so. Comparing the second assignment results (shown in Figure 2) there is a slight increase in the average mark in the checklist group, however again this result is not statistically significant ($t = 0.33$, $p > 0.05$).

	Average	**Std. Dev.**	**% Non Submission**
Control Group	51.70	14.22	22.85
Checklist Group	56.98	17.59	57.89

Figure 2. Follow up study: Checklist group vs. Control group

Despite using a cohort comprising two courses over two years the numbers are still small with 22 submissions for the first group and 16 for the second. The code review exercise at the end of the first assignment provides a larger sample size. Both groups of students in the two years of the follow-on study were surveyed after undertaking the first assignment, and 46 of a total of 50 students who submitted assignment one completed the survey. At this point all students either through the first assignment or the laboratory sessions should have undertaken the code review process. The results show that the majority of students carried out the code review (87%) and believed it had improved the quality of their programs (80%). The results indicate that the students had a much better understanding of the points on the code review (68%), that it was well structured (58%) and that they understood the terminology (77%). The majority of students felt that code reviews were useful in learning to write good programs (95%) and would use them in future to improve their marks (83%).

The students felt the best things about the code reviews were that they provided an opportunity to strategically check the code for errors (37%) in their code and that it reminded them of things that they had forgotten (39%). The major issue reported by students in terms of using the code review was the time required to undertake (29%) though a few students indicated problems with the terminology (13%) and its application (7%). Students struggled to undertake the code review for a variety of reasons top of these being a lack of time (40%). Some reported problems with the terminology (19%) and others the application of the process (19%).

5. DISCUSSION

Research has demonstrated that code reviews can improve the quality of the programs produced by professional programmers (Sommerville, 2007b), groups of students (Hundhausen et al, 2009; Trytten, 2005; Wang et al, 2008) and individual programmers as part of a highly disciplined process (Humphrey, 1997). Regarding an improvement in performance the results are inconclusive, though a small increase is shown in both studies this is statistically insignificant. Overall the survey results suggests that the students who undertook the code review thought the process was beneficial and believed that undertaking the review had improved the programs they had produced.

Potentially the students corrected minor omissions and errors which the marking scheme was not sensitive enough to detect. Some students reported being reminded of what they had forgotten while others gained knowledge though the checklist did not introduce any new material. A number of students indicated that the code review provided a mechanism of structured review of their programs and encouragingly the majority of students reported they would use code reviews for future assignments.

The high percentage of non-submissions checklist groups is unexplained but may have impacted on the results. The group selection in the pilot study produced two groups with a similar profile based on the previous assignment, and both groups were subject to the same external pressures. There were students who did not undertake the code review. This is unsurprising in the pilot study as there were no marks directly associated with the completion of the code review. This suggests that some of the first year students lack the discipline required to undertake the review, a formal structure such as that of the PSP approach (Bloch, 2000, Humphrey, 1997) may be required. The situation was improved in the follow-up study with the allocation of 5 marks for the completion of the code review in the second assignment, however only 52% of students made use of the code review. Time seems to be a critical factor in terms of student interaction with the code review, time required to complete the review being highlighted as a key drawback. Students who overlooked the code review while focusing on the code may have been running out of time.

The results of the survey and the observations of staff after the pilot suggest that a number of students found the checklist difficult to apply, some were unsure as to how to apply the code review, whether it should be a continuous process or something which only takes place at the end. The terminology used in the checklist was also an issue, highlighted both in the survey and by staff, unfortunately the points in question were not recorded. The follow-up study appears to have resolved these issues with the incremental introduction of the checklist; this is reflected in the survey data from the follow-up study and the observations of staff.

During the pilot study staff members were also informed that students from the control group had obtained copies of the code review. There were also students in the first year of the follow-up study (who were not asked to review their code for the second assignment) who made use of the code review in order to get a better mark. This is encouraging and these keen students are a credit to our university but their actions are damaging in terms of our results.

6. CONCLUSION

In conclusion it remains unclear whether code reviews can be used as an additional component to first year assignments to improve program quality. Perhaps as software engineering texts suggest quality cannot be built in at the end (Sommerville, 2007b). However from a pedagogical perspective students have found the process beneficial. They feel their programs have improved suggesting they have found errors and omissions in their code and corrected them. The code reviews have served to remind students of good practice, provide them with additional information and most importantly allow them to strategically review their programs. An analogy can perhaps be made with proofreading. Observations suggest only those students who finish their reports in good time will proofread. This has also been observed in other studies (Gambell, 1991). Similarly few students will proofread their code to identify errors/omissions. Once the code compiles and passes rudimentary testing it is submitted. Code reviews may provide a mechanism for encouraging students to proofread code prior to submission. A larger future study is required to ascertain whether code reviews increase student performance.

There were a number of improvements in the follow-up experiment which are highlighted for future studies, notably the code review formed part of the submission with associated marks. This provided a crude incentive for students to undertake the review. In the future the submission will also include some method for students to indicate which elements they found useful and anything they did not understand, providing a mechanism for improving the review in future years. Work needs to be undertaken to make the code review process as simple as possible for the student, for example, the use of a template reporting table. Developing the code review checklist on a topic by topic basis provided students with a reference for best practice and a review checklist for their assignments. The checklist points could be graded starting with syntax and moving towards design considerations as in earlier work (Trytten, 2005).

It has been noted that peer and individual assessment in higher education requires training and academic maturity amongst the students (Fallows and Chandramohan, 2001). If exposing students to the code review process early in the year can provide them with a framework for assessing their code and making improvements, it may be possible to extend the checklist into a means for self or peer assessment. Further work is required to evaluate code review as a means for self and peer assessment.

7. REFERENCES

Armitage, A., Bryant, R., Dunnill, R., Flanagan, K., Hayes, D., Hudson, A., Kent, J., Lawes, S. and Renwick, M. (2007) *Teaching and the Management of Learning* In *Teaching and Training in Post-Compulsory Education* Open University Press, Berkshire, England, pp 88-142.

Aybuke, A., Petersson, H. and Wohlin, C. (2002) State-of-the-Art: Software Inspections after 25 Years*, Software Testing, Verification and Reliability,* 12,(3) pp 113-154.

Bagel, A. and Nachiappan, N. (2008) Presented at *2nd ACM-IEEE International Symposium on Empirical Software Engineeirng and Measurement*, Kaiserslautern, Germany, pp.

Bloch, S. A. (2000) Scheme and Java in the First Year*, Journal of Computing in Small Colleges,* 15,(5) pp 157-165.

Boehm, B. W. (1979) *Software Engineering; R & Dtrends and Defense Needs* In *Reserach Directions in Software Techology*(Ed, Wenger, P.) MIT Press, Cambridge, MA, pp 22:1-9.

Brunner, J. S. (1985) *Vygotsky: A Historical and Conceptual Perspective* In *Culture, Communiation and Cognition, Vygotskian Perspectives.* (Ed, Wertsch, J. V.) Cambridge University Press, Cambridge, pp 21-34.

Brykczynski, B. (1999) A Servey of Software Inspection Checklists*, ACM SIG SOFT Software Engineering Notes,* 24,(1) pp 82-89.

Chapparro, E. A., Yuksel, A., Romero, P. and Bryant, S. (2005) Presented at *17th Annual Workshop Psycology of Programming Interest Group*, Brighton, Sussex, UK, pp 5-18.

Doyle, J. K. (2005) Improving Performance and Retention in Cs1*, Journal of Computer Science in Colleges,* 21,(1) pp 11-18.

Edwards, S. H. (2003a) Improving Student Performance by Evaluting How Well Students Test Their Own Programs*, Journal on educational resources in computing,* 3,(3) pp Article 1.

Edwards, S. H. (2003b) Presented at *18th International Conference on Object Orientated Programming Systems, Languages and Appilcations*, Anaheim, CA, USA, pp 148-155.

Fagan, M. E. (1976) Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Jounral,* 15,(3) pp 182-211.

Fallows, S. and Chandramohan, B. (2001) Multiple Approaches to Assessment: Reflections on the Use of Tutor, Peer and Self Assessment*, Association for Learning Technology Journal (ALT-J),* 9,(1) pp 26-37.

Gambell, T. J. (1991) University Education Students' Self-Perceptions of Writing*, Canadian Journal of Education,* 16,(4) pp 420-433.

Huggins, J. K. (2009) Engaging Computer Science Studetns through Cooperative Education*, ACM SIGCSE,* 41,(4) pp 90-94.

Humphrey, W. S. (1997) *Finding Defects* In *Introduction to the Personal Software Process* Addison Wesley Longman Inc, New York, USA, pp 157-174.

Hundhausen, C., Agrawal, A., Fairbrother, D. and Trevisan, M. (2009) Intergrating Pedagogical Code Reviews into a Cs 1 Course: An Emperical Study*, ACM SIGCSE,* 41,(2) pp 291-295.

Jacobson, N. and Schaefer, S. K. (2008) Pair Programming in Cs1: Overcomming Objections to Its Adoption*, ACM Special Interest Group on Computer Science Education,* 40,(2) pp 93-96.

Ma, L., Ferguson, J., Roper, M., Wilson, J. and Wood, M. (2005) Presented at *6th Annual Conference of the ICS HEA*, York, UK, pp.

McDowell, C., Werner, L., Bullcock, H. E. and Fernland, J. (2002) Presented at *ACM Special Interest Group on Computer Science Educaton Technical Symposium*, Cincinnati, Kentucky, pp 38-42.

McDowell, C., Werner, L., Bullcock, H. E. and Fernland, J. (2006) Pair Programming Improves Student Retention, Confidence and Program Quality*, Communications of the ACM,* 39,(8) pp 90-95.

Muller, M. M. (2005) Two Controlled Experiments Concerning the Comparison of Pair Programming and Peer Review*, Journal of Systems and Software,* 78,(2) pp 166-179.

Panselinas, G. (2009) 'Scaffolding' Though Talk in Groupwork Learning*, Thinking Skills and Creativity,* 4 pp 86-103.

Petty, G. (2004a) *Group Work and the Art of Student Talk* In *Teaching Today* Nelson Thrones, Cheltnam, UK, pp 218-233.

Petty, G. (2004b) *Learning from Experience* In *Teaching Today* Nelson Thrones, Cheltnam, UK, pp 319-329.

Phongpaibul, M. and Boehm, B. (2006) Presented at *ACM-IEEE International Symposium of Empirical Software Engineering*, Rio de Janerio, Brazil, pp 85-94.

Race, P. (2005) *Beyond Learning Styles?* In *Making Learning Happen* Sage Publications, London, pp 42-64.

Reece, I. and Walker, S. (2007) *Assessment of Learning and Achievement* In *Teaching, Training and Learning: A Practical Guide* Business Education Publishers Ltd., Tyne and Wear, UK, pp 321-384.

Sheard, J., Simon, S., Hamilton, M. and Lonnberg, J. (2009) Presented at *5th International workshop on Computing education research*, Berkeley, CA, USA, pp 93-104.

Simon, B. and Hanks, B. (2008) First-Year Students' Impressions of Pair Programming in Cs1*, Journal on educational resources in computing,* 7,(4) pp 5:1-5:28.

Sommerville, I. (2007a) *Software Process* In *Software Engineering* Pearson Education Limited, Essex, England, pp 65-91.

Sommerville, I. (2007b) *Verification and Validation* In *Software Engineering* Pearson Education Limited, Essex, England, pp 516-136.

Trytten, D. A. (2005) A Design for Team Peer Code Review*, ACM SIGCSE,* 37,(1) pp 455-459.

Turner, S. A., Quintana-Castillo, R., Perez-Quinones, M. A. and Edwards, S. H. (2008) Presented at *39th Special Interest Group on Computer Science Education*, Portland, OR, USA, pp.

Wang, Y., Li, Y., Collins, M. and Liu, P. (2008) Process Improvement of Peer Code Reivew and Behaviour Analysis of Its Participants*, ACM SIGCSE,* 40,(1) pp 107-111.

Williams, L. (2007) Lessons Learned from Seven Years of Pair Programming at North Carolina State University*, ACM Special Interest Group on Computer Science Education Bulletin,* 39,(4) pp 79-83.

Williams, L. and Upchurch, R. L. (2001) In Support of Student Pair-Programming*, ACM SIGCSE Bulletin,* 33,(1) pp 327-331.

Williams, L. A. and Kessler, R. R. (2000) All I Needed to Know About Pair Programming I Learned at Kindergarten*, Communications of the ACM,* 43,(4) pp 108-114.