Experiences Using Defect Checklists in Software Engineering Education

Kendra Cooper[1], **Sheila Liddle**[1], **Sergiu Dascalu**[2]

[1]Department of Computer Science
The University of Texas at Dallas
Richardson, TX, 75083 USA
{kcooper, skl031000}@utdallas.edu

[2]Department of Comp. Sc. & Engineering
The University of Nevada, Reno
Reno, NV, 89557 USA
dascalus@cse.unr.edu

Abstract There are numerous challenges in teaching software engineering courses, as such courses typically cover multiple technical, managerial and social topics. Within software engineering, software quality assurance (SQA) is a complex area to teach, because it involves aspects from all these three types of topics. Given the complexity of the area and the limited amount of time available to teach a software engineering course educators need to ask the question "*How can we effectively teach some of the most important SQA issues and techniques, highlighting the critical relationship between quality and the time and cost of software development?*". Here, we present a survey of the literature on checklists and our experiences using defect checklists in software engineering classes at both graduate and undergraduate levels. The study involves 11 teams, who used the checklists to collect defect data on technical deliverables, in both internal (peer) reviews and external (instructor) reviews. Using an iterative waterfall model, students were required to correct the defects discovered in internal and external reviews; the purpose of this was to reinforce the relationship between the quality of the deliverables and the time and effort required for correcting defects. The defect data are analyzed with respect to five conjectures; the strengths and limitations of the approach are discussed using the results. Improvements to the approach and alternatives are suggested, which could aid educators in teaching this important area of software engineering.

Keywords: software engineering education, software quality assurance, defect checklists

1. Introduction

Promoting software quality is a challenging task in teaching software engineering (SE) courses. One critical aspect of educating students in evaluating the quality of a SE artifact or process is to provide timely feedback on their deliverables. The feedback is essential for the students to understand where the models have high quality, where they need to be improved and, most importantly, why the models are considered of high quality or not.

At the University of Texas at Dallas (UTD), in CS/SE 6354 Advanced Software Engineering (graduate) and CS/SE 3354 Software Engineering (undergraduate) software engineering courses, there are typically 10 deliverables for the students to complete and the instructor to review, not once, but iteratively. For example, when the analysis model is delivered, it needs to be reviewed both internally and with respect to the requirements model for consistency, correctness, clarity, and completeness. The course deliverables include two plans (project management plan and software quality assurance plan), a set of technical models (requirements use case model, analysis model, architecture model, detailed design model, implementation solution, and a set of system level blackbox test cases), a team project website, and a project demonstration.

Originally, CS/SE 6354 was taught without the use of checklists. As reviews occurred over the terms, patterns of common mistakes emerged in the external review process. There were similar mistakes across teams within one section and across multiple sections of the course. The high number of defects required a great deal of time and effort to provide feedback to the students and the long lists of review comments were demoralizing for the students. Consequently, defect checklists for the technical deliverables were prepared to try to capture the expertise of the external reviewer and make it accessible to the students. Currently, more than 50 defect checks are included in the lists for the requirements, architecture, analysis, detailed design, implementation, and blackbox system level test cases. Now that the checklists are available, students have been asked to use the checklists both while they were developing the models and in their internal peer review. The goals were to have the students: (a) use the checklists while they developed the models to remove defects before the internal review, and (b) use the checklists as part of the internal review process to remove defects before the external review occurred. The results were significant, as in general the quality of the deliverables improved dramatically and, consequently, the time and effort to perform the external reviews was reduced. Later, when teaching CS/SE 3354, the checklists were used immediately, with similar positive results.

This paper provides a report on our practical experiences with using checklists in SE, specifically as applied in the Department of Computer Science at the University of Texas at Dallas. The approach is original in that it is implemented rather extensively in an academic environment, where significant challenges exist, in particular: short intervals between deliverables, lack of experience by students, and demanding requirements to process and interpret the checklists within short periods of time. It is also original through the specifics of the checks used, all carefully refined by the authors to suit the needs of both graduate and undergraduate one-semester SE courses. As presented later in the paper, through detailed observations and analysis of results various positive aspects of using checklists for developing software projects and teaching SE in academia have been identified. In addition, several limitations of the checklist-based approach are now better understood, a necessary premise for finding adequate solutions to address them. Based on the work done at UTD, a collaboration has been initiated with the Computer Science and Engineering at the University of Nevada, Reno (UNR), the idea being to extend the application of checklist-based approach to SE courses taught at other universities and, through extended participation, refine the approach's principles, modes of application, and specific details. Thus, the experience report presented in this paper provides the foundation for developing a more comprehensive, refined work plan, to be applied in future projects, including in inter-university collaborations such as the one started by the authors of this paper. In short, the main goal of the proposed approach is to lead to a practical, more general solution to teaching SQA as part of SE education and instill a "discipline of software verification" (leading to high quality software products) in new generations of students who take SE courses.

This paper, in its remaining part, is organized as follows: Section 2 surveys related SQA work, in particular the use of checklists and their alternatives. The checklists used in this study are presented in Section 3. An analysis of the defect data collected in the study is provided in Section 4. The experiences using the checklists are discussed in Section 5. Conclusions and future work are in Section 6.

2. Related Work in Software Quality Assurance

Checklists have been used for quite sometime in software engineering, particularly since Fagan's seminal paper on design and code inspections [13]. Their advantages stem from their simplicity, ease-of-use, cost-effectiveness, and support for thorough verification of software artifacts. In time, checklists have become comprehensive verification devices in software engineering, typically associated with the phases of the software life cycle (i.e., checklists for requirements, design, implementation, and testing), but sometime also possibly tailored to individual developers and/or projects [6]. According to the same author, checklists encompass items for non-code workproducts, general programming practices, specific programming languages, style issues, and particular projects [6].

More recently, specific categories of checklists have been developed to deal with particular concerns, such as software security [15], safety [10], and cost management [16]. Furthermore, as development techniques and tools have evolved, so did checklist-based SQA techniques. For example, object-oriented programming has been followed by a profusion of newer checklist-based verification techniques (e.g., [11]) and the relatively recent use case modeling has led to newer inspection techniques [18], [2] [19][22]. Comprehensive lists of checklists have been put together, for example in [1][3][13][17][20]; a trend accompanied by research on checklist processing and defect detect estimation (e.g., [4][21][23]).

Checklists, however, are only one type of technique to be used in software verification and validation. Based on surveying several related studies, Endres and Rombach indicate that combinations of verification and validation techniques (such as inspection, together with white box testing and black box testing) are more effective than individual solutions [12]. In terms of reading techniques alone (for inspection) Thelin *et al* point out that several alternatives to checklist-based approaches exist: defect-based, perspective-based, traceability-based, and usage-based [22]. Checklists have also been developed and used in academic settings, for example at the Software Engineering Research Lab of the University of Kaiserslautern, Germany [1]. Several software engineering textbooks cover them rather comprehensively [5][17][20]. In general, we estimate the most software engineering courses include them to certain degree (for example, at UNR, checklists have been used extensively for project demonstrations [9]). Nevertheless, the approach described in this paper is unique in that it comprehensively uses checklists throughout much of the software process, applied throughout a complete course cycle (a semester) at both undergraduate and graduate levels [7][8]. The amount of detail in each checklist, the number of checklists, their frequency of use, and the number of students involved in using them are also specific to our approach.

3. Checklists

The checklists used in this study cover the *use case model of the requirements*, *architecture model*, *analysis model*, *detailed design model*, and *blackbox system level test cases*. Additional checklists (e.g., for the project management plan, software quality assurance plan, etc.)

are available, but are not reported in this work. The checklists have been developed for students. The main goals of the checklists are that students could readily understand the meaning of each check and apply the check to a software engineering artifact without significant additional training. For illustration purposes, the requirements checklist is presented in TABLE I. The checks are tailored for object-oriented, UML models. Here, the checks have been designed to be specific; many would be straightforward to implement in a CASE tool such as, for example, checks to see if a graphic use case has a text description and each text description has a graphic use case represented in the diagram. Other checks, however, are more subjective, such as check #4 in TABLE I, which verifies that the name of the use case clearly conveys its purpose to reviewers. A check like this could not be fully automated.

Item	Defect Description
1.	Each graphic use case has a textual description
2.	Each textual description has a graphic use case
3.	Each use case is named with a verb phrase
4.	The name of use case conveys the purpose to reviewers
5.	Each text use case has actors, entry conditions, exit conditions, normal flow of events, alternate flow of events (optional), and non-functional requirements (optional)
6.	Each actor on a use case diagram is described
7.	Each concrete use case is initiated by one or more actors
8.	Events in textual description are externally visible
9.	Data elements input by the user or output by the system are clearly described (not referred to as "information" or "data")
10.	Multiplicity of data elements is described (optional, required, multiple allowed)
11.	Use cases to initialize and shutdown the application are present
12.	Use case model is not overly complicated (e.g., an abstract use case is re-used by many concrete use cases with the include feature)
13.	Each use case is approximately 1-3 pages (exceptions need to be justified in rationale)
14.	Actors represent external people or systems
15.	Rationale for the use case model is provided, well thought out
16.	Other

TABLE I. Defect checklist for the requirements (use case model)

It is worth noting that all the checklists used are evolving through constant re-evaluation and refinement. For example, since their initial use, additional checks have been identified for the requirements use case model. Such additions include: (a) Each abstract use case is initiated by one or more use cases; (b) Each actor, or its generalization, triggers a concrete use case; (c) The abstract use case, which extends another use case, clearly identifies where the former intercedes, and (d) The abstract use case, which uses another use case, clearly identifies where the latter is invoked.

4. Evaluation of the Defect Data Collected

4.1 Organization of the evaluation

The evaluation of the defect data collected is organized with respect to a set of conjectures, which are proposed based on observations made while teaching software engineering courses both with and without using defect checklists. It is important to note, however, that the conjectures are based on subjective observations, as no data has been collected in the past about the defect metrics. Based on the results, the conjectures will be re-examined, updated, and additional studies will be conducted. Ultimately, validated conjectures can be used to improve software engineering education. For example, if it is the case that the requirements consistently have high defect rates with the current course organization, then additional lecture time or tutorials can be organized to improve the quality of the requirements. Additional conjectures and analysis work are available, but not reported here due to space restrictions.

Conjecture 1. The total number of defects discovered in each undergraduate teams is within +/- 10% of the average number of defects found across the teams (i.e., the total number of defects in each team is approximately the same). The assumption here is that given the same project, about the same number of defects will be introduced into the artifacts by the undergraduate students.

Conjecture 2. The total number of defects discovered in each graduate teams is within +/-10% of the average number of defects found across the teams (i.e., the total number of defects in each team is approximately the same). The assumption is that given the same project, about the same number of defects will be introduced into the artifacts by the graduate students.

Conjecture 3. The average total number of defects discovered in the graduate teams is higher than the average total number of defects found in undergraduate teams. The assumption is that the increased complexity of the graduate project in comparison to the undergraduate project leads to the introduction of more defects into the artifacts by the students.

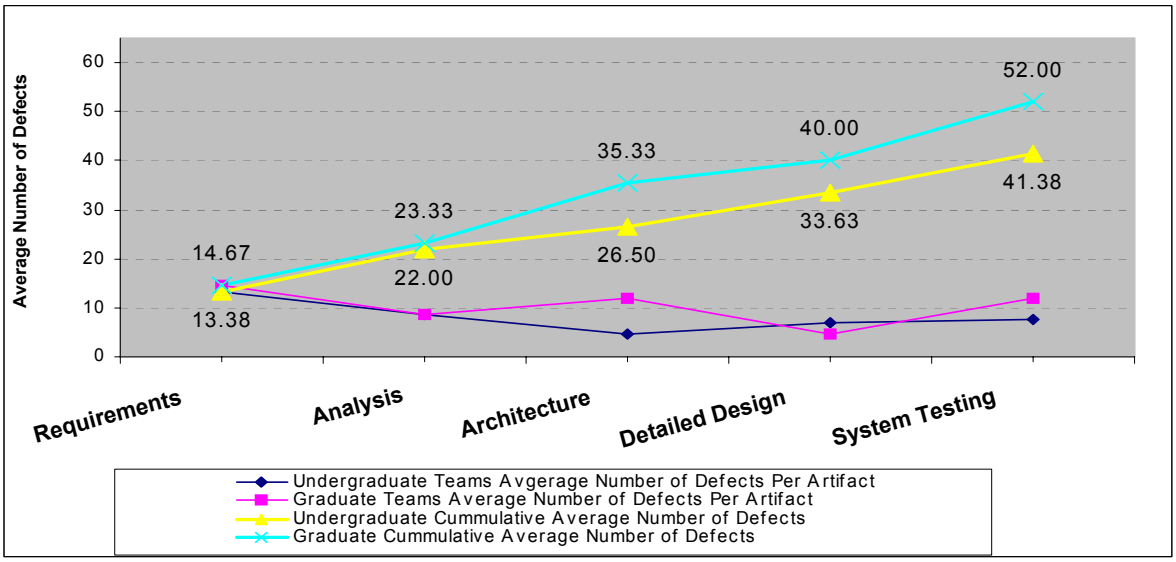

Figure 1. Exploring defect data for software engineering artifacts

Conjecture 4. The ratio of the number of defects discovered in the internal reviews by graduate teams is higher than the ratio of the number of defects discovered in the internal reviews by undergraduate teams. The assumption is that a graduate team is better prepared identify defects, as they have a richer background in computer science upon which to build software engineering skills.

Conjecture 5. The ratio of the number of defects discovered in the internal reviews by undergraduate teams to the total number of defects detected is proportional to the academic level of the team. The assumption is that a team composed of senior level students, for example, is better prepared to identify defects in comparison to a team composed of junior level students, as they have a richer background in computer science upon which to build software engineering skills.

4.2 Results of the evaluation

Defect data collected in this study, related to the above five conjectures, is available via [8]. Results for the conjectures are presented below using summaries of data in tables and a variety of plots.

4.2.1 Observations on Conjectures 1 and 2

The total number of defects discovered in each undergraduate team, graduate team, and the average number of defects found across the teams is summarized in TABLE II.

Based on the results, neither conjecture 1 or 2 is supported. A wide range of percentage differences is reported. The undergraduate teams range from approximately -44% to +32%; the graduate teams from -25% to +21%. The assumption for these conjectures needs to be re-examined (i.e., is it true that that given the same project, about the same number of defects will be introduced into the artifacts by the students). The results indicate the project itself is only one of many contributing elements in the total number of defects discovered by teams. For example, if the total number of defects is very low, then this may indicate that team members: (a) are detecting and correcting defects before the deliverables go through internal and external reviews (this may be the case for teams whose members are rigorously using the checklists in desk checks); (b) have substantial software engineering experience and are introducing fewer defects in the artifacts; or (c) are reporting (i.e., counting) defects in a more general way. For example, one team may report a problem defect that occurs in multiple places as a single defect; other teams may be reporting it as many occurrences. We also note that the range of the percentage difference calculations is larger for the undergraduate teams than it is for the graduate teams. This may be due to the smaller sample size for the graduate teams that is available for this study. Additional studies to determine the causes of the variability are needed.

4.2.2 Observations on Conjecture 3

The data collected for the total number of defects for the undergraduate and graduate teams is explored by plotting the number of defects discovered per artifact and comparing the undergraduate and graduate teams, as shown in Figure 1. The graduate teams generally have a larger number of defects detected for each artifact. The single largest number of defects is detected in the requirements artifacts for both undergraduate and graduate teams. This may be because the students are less

TABLE II. Total and average number of defects per team

Number of Defects Per Team	% Difference
Undergraduate Team Data	
23	-44.41
24	-41.99
35	-15.41
45	8.76
48	16.01
48	16.01
53	28.09
55	32.93
Total Number of Defects	331
Average Number of Defects Per Team	41.37
Graduate Team Data	
Number of Defects Per Team	% Difference
39	-25.00
54	3.84
63	21.15
Total Number of Defects	156
Average Number of Defects Per Team	52.00

familiar with developing requirements than other deliverables (e.g., at the design or code level). The relatively constant detection of defects throughout the lifecycle is also interesting to observe, which occurs in both the undergraduate and graduate teams. This supports the need for applying SQA techniques, such as checklists, throughout the development lifecycle.

4.2.3 Observations on Conjecture 4

The data collected for the ratio of defects detected internally with respect to the total number of defects for the graduate and undergraduate teams is summarized in TABLE III. The data in TABLE III is sorted on increasing number of defects. The results indicate the graduate teams have a higher ratio of internally detected than undergraduate teams; the difference is approximately 45%. These results support the conjecture that graduate student teams have a higher ratio of internally detected defects than undergraduate teams.

TABLE III. Ratio of internal to total number of defects

Internal of Number of Defects Per Team	Total Number of Defects Per Team	Ratio
Undergraduate Team Data		
1	23	0.04
5	45	0.11
6	48	0.12
10	53	0.19
15	24	0.62
15	35	0.42
20	48	0.41
30	55	0.54
Total Number of Internally Detected Defects	102	
Total Number of Defects	331	
Average Ratio	30.8%	
Graduate Team Data		
Internal/Total Ratio of Number of Defects Per Team	Total Number of Defects Per Team	Ratio
18	39	0.46
18	54	0.33
52	63	0.82
Total Number of Internally Detected Defects	88	
Total Number of Defects	156	
Average Ratio	56.4%	

4.2.4 Observations on Conjecture 5

The data collected for the relationship between the ratio of defects detected internally to the total number of defects and the academic level of the undergraduate teams are presented in a scatter plot in Figure 2. The average academic level for each team is calculated using the following weights. A freshman has weight 1, sophomore 2, junior 3, senior 4, and graduate student 5. Thus, for example, a team composed of two juniors and two seniors has an academic level (3+3+4+4)/4 = 3.5.

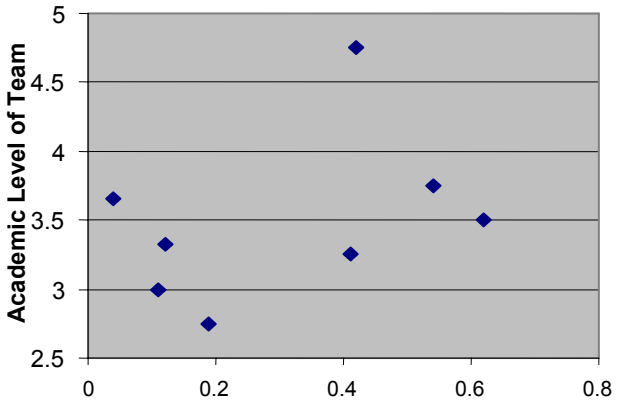

Figure 2 Exploring the relationship between academic level and internally detected defect ratios in teams

Based on the results, conjecture 5 is not supported, as an increasing relationship (i.e., the ratio of internally detected defects increases with the academic level of the team) is not evident. These results indicate many possible items to consider. For example, the underlying assumption of the conjecture (a team composed of senior level students is better prepared to identify defects in comparison to a team composed of junior level students, as they have a richer background in computer science upon which to build software engineering skills) needs to be re-examined. Other elements may have a more significant impact, for example work experience either in full time positions or co-operative education program. In addition, the three possible items that were identified for conjectures 1 and 2 may also impact this conjecture, as they deal with fundamental issues around identifying and counting defects (refer to Section 4.2.1). Additional studies to identify relationships are needed.

5. Discussion

5.1 Using a Checklist Approach

5.1.1 Strengths of the Checklist Approach

Based on the observations obtained from the educational experiments conducted over a period of an academic year (2004/2005) and the processing and interpretation of data gathered, the following have resulted as the main strengths of the proposed checklist-based SQA approach.

First, checks are specific and "atomic", with clear meaning, generally easy to evaluate, and with only possible answers "yes" or "no". These characteristics make them very suitable for application by both novice and advanced software engineers "in-training" as little additional study is required (besides the one necessary to understand other SE concepts, for example needed for requirements gathering, use case modeling, or test case generation).

Second, many checks could be embodied in CASE tools which, means that much faster solutions (than the manual ones) could be devised for completing the checklists (that is, providing answers to checks), analyzing the information collected, and presenting the results of the checklists' interpretation.

Third, as the large majority of checks are rather simple, they not only can be scanned and evaluated relatively quickly but also can provide a pragmatic "engineering device" that significantly reduces the risk of omitting important aspects of the software models checked.

Fourth, checklists are powerful in that they constitute a concrete technique that the students can use for verifying the software they are creating. Notably, in the absence of checklists, in many projects developed in both industry and academia, such verification is either ad-hoc and arbitrary, or simply non-existent.

5.1.2 Limitations of the Checklist Approach

Also based on the observations resulted from the experiments conducted and the analysis of data gathered, the following have been identified as being the main limitations of the proposed checklist-based SQA approach.

First, like any checklist-based approach, the one presented in this paper is not comprehensive. Checklists cannot cover all possible verification and validation aspects of an artifact or product (in our case, a software model) and, as pointed out by Endres and Rombach, they need to be accompanied by additional techniques [12].

Second, the checklists used in the study presented in this paper have been developed for students, not experts. Thus, the checks do not necessarily have all the four Cs (correctness, completeness, consistency, clarity) required to more formal SQA. More specifically, since they were intended to be used by students, the checks were created with emphasis on correctness (but not in a formalized, mathematical way), consistency, and clarity. Less weight was placed on completeness, which is typically a difficult goal to achieve.

Third, also because from its inception the checklist-based approach has been targeted to software engineering students, its scope is limited to educational environments and thus less likely to perform well in industry-strength software development projects. An extended, alternative set of checklists for use by experts represents a future direction for the work presented here.

Fourth, templates, guidelines, and training need to be provided to ensure a more consistent approach in identifying and recording defects. For example, if an

external review indicates the alternate flows are not defined for one use case and the comment is made to check all of the use cases for the same problem, then how should this be recorded? Ideally, if six use cases are discovered to have the same problem, then the number of defects recorded should be six. Currently, no guidelines have been provided; the approach taken by each team has been ad-hoc. In addition, each team has collected data in different formats, using different tools (Word, Excel, etc.). This diversity makes it time consuming to organize and analyze the data. Providing the students with templates for the data collection would help to solve this problem.

5.2 Software Quality Assurance Education

Software quality assurance issues seem to raise significant challenges for students. In our experience, many teams (both graduate and undergraduate) had difficulties in organizing and conducting internal peer reviews, did not adequately use the defect checklists provided by the instructors, and did not consistently collect defect data as needed for the peer review process. These observations raise interesting questions which, if properly answered, could lead to useful solutions and conclusions.

Naturally, the first question raised is "what could be the causes for the above less than expected students' involvement and results in applying SQA techniques?" Among possible explanations are: (a) the need and the importance of defect detection and correction were not sufficiently emphasized by the instructors; (b) the students had not enough experience in dealing with general project-related time and risk management issues; (c) comprehension of SQA material could be more difficult and might require more experience and scientific maturity than other areas of SE; and (d) the scope of the project was rather large, involving multiple and complex aspects of software engineering, including technical development aspects, process and project management, and sustained team collaboration (all, in the rather short timeframe of an academic semester).

Consequently, the next major question raised by the study conducted is "what practical improvements are within the instructors' reach to raise the level of SQA education?" Again, some possible answers to this question are as follows: (a) provide a well-defined SQA plan to alleviate the burden of the students when trying to integrate the material on their own (this plan should be easily adjustable to various types of team projects); (b) prepare a set of well-defined defect collection templates that encapsulate "distilled SQA experience", to be passed smoothly from the instructors to the students; (c) also related to templates, define a unified, consistent set of data collection forms (defect data was collected in diverse formats and styles, and no standard defect data collection form was provided to the students; however, this should be straightforward to build and would certainly improve the consistency of data collection in the future); and (d) provide guidelines that explain each possible defect in detail, include example(s) of each defect, and describe in detail how to record defect data in templates.

Yet another interesting question raised by the study conducted is "at what point providing templates and guidelines might become too much?" That is, where should be drawn the line between closely guiding the students in their SE work and letting them think creatively and encouraging them to propose their own solutions for collecting, analyzing and repairing software defects? Obviously, this last question has no straightforward answer and needs to be carefully considered in conjunction with further application of the proposed approach.

Furthermore, several other questions were raised by the study conducted and the analysis of data collected, for example "what is the true relationship between the students' academic level and their proficiency in applying SQA techniques?". These, as practically all the above questions, are open questions, whose possible answers, if well constructed, would certainly lead to better education in SE in general, and SQA in particular.

6. Conclusions and Future Work

The work presented in this paper is about defining and following a checklist-based approach for enhanced SQA education. Details of the approach, examples of checklists used, excerpts of data collected during a semester when the approach was applied in both a graduate and an undergraduate course, as well as an analysis and a discussion of the results obtained have been included in the paper.

Notably, the experiences reported have led to interesting and useful observations, challenging questions, and several possible answers. Most importantly, the study conducted has strengthen our belief that more needs and could be done in order to increase the level of software engineering education, for the obvious benefit of having students better prepared to create higher quality software, useful in a wide range of human activities.

In terms of future work, we plan to concentrate on proposing effective solutions to address the questions outlined in the previous section of the paper. These solutions will encompass a refined, better-designed and more comprehensive SQA plan, a more complete and standardized set of templates for recording and repairing defects, and a set of more detailed quidelines for using the SQA plan and its associated checklists. While refining the approach, we will also update the conjectures. Ultimately,

validated conjectures can be used to improve software engineering education. For example, if it is the case that the requirements consistently have high defect rates with the current course organization, then additional lecture time or tutorials can be organized to improve the quality of the requirements. Furthermore, we would like to extend the application of the proposed approach and its related "devices" (plan, templates, guidelines) to new environments (other universities) and new projects, including multi-disciplinary software-intensive projects.

7. References

[1] Software Engineering Research Lab (AGSE) website at the University of Kaiserslautern, Germany, Checklist Inspection of Use Cases, accessed June 20, 2005 at http://wwwagse.informatik.uni-kl.de/teaching/se1lab/ss2003/SysAnfBearbeiten/ChecklistenMitPerspektiven_Analyse.pdf

[2] Anda, B. and Sjøberg, D.I.K., "Towards an Inspection Technique for Use Case Models," *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* (SEKE'02), pp. 127-134.

[3] R.S. Pressmann & Associates, Inc., APM Document Template website, "Adaptable Process Model – Software Engineering Checklists," accessed June 20, 2005 at http://www.rspa.com/checklists/index.html

[4] Biffl, S. and Grossmann, W., "Evaluating the Accuracy of Defect Estimation Models Based on Inspection Data From Two Inspection Cycles," *Proceedings of the 23rd International Conference on Software Engineering (ICSE-2001),* pp. 145-154.

[5] Bruegge, B. and Dutoit, A.H., *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 2nd Edition, Pearson Prentice Hall, 2004.

[6] Brykczynski, B., "A Survey of Software Inspection Checklists," *ACM Software Engineering Notes*, 24(1), January 1999, pp. 82-89.

[7] Cooper, K., Course syllabus CS/SE 3354 Software Engineering, Spring 2005 at: www.utdallas.edu/~kcooper/teaching/3354/3354spring05/3354spring05.htm

[8] Cooper, K., Course syllabus CS/SE 6354 Advanced Software Engineering, Research Track, Spring 2005, at:www.utdallas.edu/~kcooper/teaching/6354/6354spring05/6354spring05.htm

[9] Dascalu, S.M., Varol, L.Y., Harris, F.C., and Westphal, B.T., "Computer Science Capstone Course Senior Projects: From Project Idea to Prototype Implementation," accepted at the IEEE Frontiers in Education Conference (FIE-2005), Indianopolis, IN, October 2005.

[10] de Almeida, J.R., Jr., Camargo, J.B., Jr., Basseto, B.A., and Paz, S.M., "Best Practices in Code Inspection for Safety-Critical Software," *IEEE Software*, 20(3), May-June 2003, pp. 56-63.

[11] Dunsmore, A., Roper, M., and Wood, M., "The Development and Evaluation of Three Diverse Techniques for Object-Oriented Code Inspection," *IEEE Transactions on Software Engineering*, 29(8), August 2003, pp. 677-686.

[12] Endres, A. and Rombach, D., *A Handbook of Software and Systems Engineering,* Pearson Education Limited, 2003.

[13] Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, 15 (3), 1976.

[14] Gilb, T. and Graham, D., *Software Inspection*, edited by Susannah Finzi, Reading, MA: Addison-Wesley, 1993.

[15] Gilliam, D.P., Wolfe, T.L., Sherif, J.S., and Bishop, M., "Software Security Checklist for the Software Life Cycle," *Proceedings of the 12th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03)*, pp. 243-248.

[16] Jørgensen, M., and Moløkken, K., "A Preliminary Checklist for Software Cost Management," *Proceedings of the 3rd IEEE International Conference on Quality Software (QSIC'03)*, pp. 134-140.

[17] Lauesen, S., *Software Requirements: Styles and Techniques*, Pearson Education Limited, 2002.

[18] McBreen, P., Software Craftmanship Inc.: Use Case Inspection Checklist website, accessed June, 2005: http://www.mcbreen.ab.ca/papers/QAUseCases.html

[19] Metropolitan Design and Development, Inc. website, Inspection Checklist for Use Case Documents file, accessed June 20, 2005 at http://www.metro-design-dev.com/files/usecasecheck.doc

[20] Pfleeger, S.L., Hatton, L., and Howell, C.C., *Solid Software*, Prentice-Hall, 2001.

[21] Porter, A.A. and Votta, L.G., "An Experiment to Assess Different Defect Detection Methods For Software Requirements Inspections," *Proceedings of the 16th International Conference on Software Engineering (ICSE-1994)*, pp. 103-112.

[22] Thelin, T., Runeson, P., and Wohlin, C., "An Experimental Comparison of Usage-Based and Checklist-Based Reading," *IEEE Trans. on Software Engineering*, 29(8), August 2003, pp. 687-704.

[23] Wohlin, C. and Runeson, P., "Defect Content Estimations from Review Data," *Proceedings of the 20th International Conference on Software Engineering (ICSE-1998)*, pp. 400-409.