A Formal Approach to Component-Based Software Engineering: Education and Evaluation

Murali Sitaraman
Dept. Computer Science
Clemson University
Clemson, SC 29634-0974, USA
+1 864 656 3444
murali@cs.clemson.edu

Timothy J. Long
Bruce W. Weide
Dept. Computer & Info. Science
The Ohio State University
Columbus, OH 43210, USA
+1 614 292 5813
{long, weide}@cis.ohio-state.edu

E. James Harner
Liqing Wang
Dept. Statistics
West Virginia University
Morgantown, WV 26506, USA
+1 304 293 3607
{jharner, lwang2}@stat.wvu.edu

ABSTRACT
This paper summarizes an approach for introducing component-based software engineering (CBSE) early in the undergraduate CS curriculum, and an evaluation of the impact of the approach at two institutions. Principles taught include a modular style of software development, an emphasis on human understanding of component behavior even while using formal specifications, and the importance of maintainability, as well as classical issues such as efficiency analysis and reasoning. Qualitative and quantitative evaluations of student outcomes and end-to-end changes in student attitudes show mostly positive results that are statistically significant, confirming that (1) it is possible to teach CBSE principles without displacing "classical" principles usually taught in introductory courses, (2) students can understand and reuse formally-specified components without knowing their implementations, and (3) student attitudes towards software engineering can be altered in directions heretofore often assumed to be difficult to achieve.

Keywords
components, formal specifications, objects, reasoning, software engineering, statistical evaluation

1 INTRODUCTION

The increasing importance of introducing component-based software engineering principles early in introductory undergraduate computer science education is widely recognized. While there is still considerable debate among educators on the principles to be taught, there is reasonable consensus that students should be educated (at least) on the following fundamental software development principles:

- *Independent software development*: Large software systems are necessarily assembled from components developed by different people. To facilitate independent development, it is essential to decouple developers and users of components through abstract and implementation-neutral interface specifications of behavior for components.
- *Reusability*: While some parts of a large system will necessarily be special-purpose software, it is essential to design and assemble pre-existing components (within or across domains) in developing new components.
- *Software quality*: A component or system needs to be shown to have desired behavior, either through logical reasoning, tracing, and/or testing. The quality assurance approach must be modular to be scalable.
- *Maintainability*: A software system should be understandable, and easy to evolve.

We have taught the above component-based software engineering principles in introductory CS education, spanning multiple courses, at The Ohio State University and West Virginia University[1] for the past several years. The technical foundations of our CBSE approach are based on RESOLVE [1]. The approach is highlighted by a "systems thinking" philosophy and formal specifications of component behavior to facilitate independent software development. It uses "design-time relationships" to capture elements of reusability and software composition, and emphasizes both formal reasoning and testing to assure high quality. (These principles are more fully described in the next section.) Introduction of CBSE principles early in the CS curriculum and the particular approach that we have used naturally raise a few basic questions:

- Is it possible to teach new CBSE principles without displacing concepts such as efficiency analysis, dynamic storage management, and recursion that are typically taught in introductory courses?
- Is it possible to teach such topics as formal specification and sophisticated reuse techniques to (early) undergraduate CS students in a way that students can understand and appreciate the significance of the principles?
- Is it possible to effect significant changes in student attitudes about software engineering?

The contribution of this paper is in addressing and answering these questions, for the most part in the affirmative. Section 2 summarizes the essential and novel aspects of our approach and educational philosophy. Section 3 explains our educational approach and provides a listing of major topics covered in the courses to address the

[1] Sitaraman was with the Department of Computer Science and Electrical Engineering at West Virginia University, while teaching and evaluating the CBSE approach presented in this paper.

0-7695-1050-7/01 $10.00

first question. It also discusses how new and traditional principles are taught in a seamless fashion. Section 4 contains summary results from qualitative and quantitative student evaluations that we administered between 1996 and 1999 to study the impact of our educational approach. It addresses the second and third questions raised above. The last section includes a discussion of lessons we learned in this process and our conclusions.

2 A SUMMARY OF TECHNICAL FOUNDATIONS

This section provides a summary of the salient aspects of our CBSE approach, and explains how they are directly motivated by general software engineering principles of independent software development, reusability, and quality assurance. These principles are also the central ones we try to communicate to our students.

To facilitate independent development of components and systems, it is essential to separate the "outside" or "abstract" view of the user of a system from the "inside" or "concrete" implementation-oriented view of the system. This is the essence of the systems thinking philosophy. For systems thinking to work, it is crucial that the abstract user-oriented descriptions of systems contain only necessary and sufficient information. To be precise, the abstract descriptions must be stated in the language of mathematics. Otherwise the very intent of having the abstract descriptions—to minimize communication overhead and to eliminate miscommunication among developers and users of systems—will be compromised.

Inevitably, all large systems must be assembled from components. In the process of designing any big system, it may be necessary to instantiate or extend previously designed components or to build new ones. To understand a system of components and to ease maintenance, it is therefore essential to understand and document the "design-time" relationships among participating components.

A key benefit of CBSE is that it permits a modular approach to ensuring software quality. Modularity is essential for the reasoning process to be scalable. In a modular or compositional reasoning approach, it possible to verify the correctness of an implementation of one component at a time using only the specifications of other components upon which it depends; implementations of reused components are not needed. Modular reasoning, of course, demands that each component *C* in the system have an abstract behavioral specification so that internal details of its use in composition can be ignored when reasoning about systems built using *C*. The property of modular reasoning requires (in addition to its reliance on specifications) careful software design.

Component specifications are also useful for specification-based integration and regression testing. In our approach, we define one-way checking (or pre-condition checking) and two-way checking (pre- and post-condition checking) components that correspond to each component in the system. Most mechanical aspects of the checking components can be generated automatically, though non-trivial condition checking, in general, requires explicit programming. The checking components help classify errors as either arising from bad components or bad client usage, i.e., they help assign responsibility for defects. By systematically employing and removing checking versions of components, it becomes possible to detect internal interface-related errors among components. In summary, the following are the central elements of our approach to CBSE:

- *systems thinking*, i.e., developing a "systems" view of software in which components can be viewed from the outside as indivisible units, or from the inside as compositions of other such systems (a.k.a. subsystems);
- *interface specifications using mathematical modeling*, i.e., not settling for mere qualitative descriptions of behaviors of components, but creating unambiguous and implementation-neutral formal descriptions using mathematics.
- *design-time relationships*, explicitly and precisely describing design-time behavioral relationships between (new and reused) components, including implementation, instantiation, extension, and composition of parameterized (template) components;
- *modular reasoning*, i.e., reasoning about client usage based on specifications, and on specification-based substitutability properties; and
- *interface violation testing*, i.e., developing test plans from specifications and using violation-checking component wrappers to detect and isolate bugs in a modular fashion.

While there are naturally differences among researchers on which principles should be considered core CBSE principles, and differences among educators on which principles ought to be taught in introductory CS classes, it is clear that there is a significant overlap between the above principles and principles suggested by other authors [9]. This is because the central tenets of software engineering, outlined in the introduction, remain the same. Related discussions may be found in [2, 3, 4, 9].

3 THE EDUCATIONAL APPROACH

The CBSE principles have been taught, using variations of RESOLVE as the delivery vehicle [1], beginning in the early 1990's at The Ohio State University (OSU) and at West Virginia University (WVU). At both institutions, instructors employed active learning approaches to teaching. This section discusses particulars of the different course sequences used at the two institutions to communicate CBSE principles. It highlights how other universities can adapt and teach CBSE principles in a way that best fits their institutional needs.

The CBSE principles have been taught in the first three-quarter sequence of undergraduate courses in CS at OSU, using RESOLVE/C++, a disciplined version of C++, as the programming language [5]. The first course provides an introduction to CBSE concentrating on the client perspective in reusing previously-developed components based on their behavioral specifications. The second course focuses on using, then at mid-term switches to implementing, generic abstract data types such as stacks, queues, lists, and maps. A typical implementation in this course is layered by reusing one or more other components. The third course focuses on completing a non-trivial component-based system using components developed in the previous courses as well as new components.

At WVU, the principles have been taught in the second semester course in CS, and on a selective basis in the junior-level software engineering course. In the second-semester course, an annotated version of Ada with RESOLVE-style specifications was used. The sections of this course examined in the surveys in this paper used RESOLVE specifications directly, with Ada reserved for lab work only. The first half of the course focused on the client perspective and the second half was devoted to implementation of abstract data types. The course emphasized specifications and specification-based reasoning [6]. In the software engineering course, principles were taught directly using RESOLVE. In addition to learning principles of requirements analysis and design, students also implemented a component-based system using components with behavioral specifications.

A summary of topics covered in the courses at OSU and WVU is given below. The listing of topics suggests how classical concepts of efficiency analysis, dynamic memory management, etc., can be taught in a component-based setting. For example, efficiency analysis techniques are introduced to compare alternative implementations of the same specification. Pointers and dynamic storage issues are discussed later as techniques for implementing "unbounded" data abstractions, and as a "from scratch" alternative to layered implementations. Loops and recursion are used to motivate inductive reasoning about correctness, and the importance of termination arguments. More details about these courses appear in [7, 8].

First Quarter: Component-based software from client programmer's perspective; intellectual foundations of software engineering; mathematical modeling; specification of object-oriented components; layering; recursion; testing and debugging layered operations.
Second Quarter: Templates for generalization and decoupling; container components; component-based software from implementer's perspective; data representation using layering and using pointers.
Third Quarter: Tree and binary tree components and binary search trees; context-free grammars; tokenizing, parsing, and code generating components; sorting components and sorting algorithms.

Figure 1: Course Descriptions for the OSU Courses

Second Semester: Principles of object-based software engineering including specification, design, implementation, and reasoning. Components developed and used in the course include generic, reusable data abstractions such as queues, stacks, lists, trees, and sorting and searching. Comparison of implementation techniques; analysis of efficiency; dynamic allocation; recursion.
Fourth Semester: Techniques and methodologies for software engineering, including principles of requirements analysis, specification, design, implementations and quality assurance. The emphasis will be on rigorous object-based software construction.

Figure 2: Course Descriptions for the WVU Courses

Some principles such as classifying and decoupling design-time relationships receive more emphasis at OSU, whereas formal reasoning was emphasized more (in some sections) at WVU. We expect such differences to arise naturally when CBSE principles are taught at different institutions.

The feedback from students confirms that they learn and see the importance of both classical topics and new CBSE principles in these courses. This is seen, for example, from Table 1 that contains a summary of top responses from students, when asked to list the most important principles learned in CS2 at the end of that course at WVU. (19 students responded to the question.) Only 3 students directly mentioned efficiency analysis, choosing instead to use terms such as "best algorithm" or "multiple implementation trade-off". We do not present any detailed analysis of this feedback, which has been used mainly in revising and improving course materials. The results indicate student perceptions in one representative section, and are not sufficient to enable us to draw any general conclusions.

Table 1: Key Principles Learned in CS2 at WVU

Principle	**Number of students who claimed they learned and understood importance**
Component-based reuse	15
Specifications	11
Dynamic storage management	8
Specification-based reasoning	6
Specific ADTs, (e.g., lists)	6
Recursion	5

4 EVALUATION

To evaluate the impact of the courses on student learning and attitudes, under grants from FIPSE (U.S Department of Education) and the National Science Foundation we administered a series of voluntary student surveys between 1996 and 1999. Three different kinds of instruments were used, in addition to regular homework/lab assignments and examinations. The first kind was concerned with providing feedback to us on how the material could be revised and better presented in subsequent course offerings. The second kind used essay questions administered near the ends of the courses to help the students tell in their own words what principles they thought they learned and what they believed were important principles for software construction. The third instrument focused on student attitudes. These questions were asked of the students at the beginning of each of the courses in the sequence and at the end of the last course. The objective was to evaluate whether student attitudes about various matters changed according to our pre-survey hypotheses. Summaries of the latter two surveys, and our interpretations of the results, follow.

Qualitative Evaluation

For the qualitative evaluation, we asked students to provide written responses to the following questions at the end of the courses:

1. List and briefly explain what you believe to be the three most crucial aspects of designing and building good software. And, which, if any, of these aspects would you have listed prior to taking the course sequence?
2. Compare and contrast your view of what a software system "should look like" with your view before taking this course sequence. If possible, organize your answer into before and after sections.

The student responses were not identified in any way to the instructors; students knew that their answers to the questions would have no effect on their grades. Though we collected responses from a number of sections, analysis of the data has proved time consuming, and we are able to present summary results from just two sections of two courses, one each at OSU (36 students) and WVU (19 students). At WVU, the questions were only asked at the end of CS2, not at the end of the sequence. These results are indicative of possible trends, and by themselves, do not confirm any hypothesis. Their likely best use for us and for others is to suggest questions that should be asked explicitly on new quantitative survey instruments in the future. Tables 2-5 contain summary results from student responses at OSU and WVU.

The OSU summary indicates more students listed design, reusability, modularity as a crucial aspect of software construction at the end of the sequence, compared to the beginning. The WVU summary indicates more students felt at the end that it was important to design/specify before coding and that it was important to reason/test to ensure software behaves as intended. More interesting trends are seen in analyzing responses to the second question. At OSU, 19 out of 24 students (whose responses could be classified) changed from having no view or a view of software as monolithic code with some use of procedures and functions, to a component-based view. At WVU, 15 out of 19 students at the end of CS2 reached a view of a software system that involved components with specifications.

While essay-type questions are costly to analyze, they can reveal interesting student perceptions. For example, at WVU, 3 out of 19 students made it clear in their responses that they already held a component-based software system view and claimed that the course had no impact on them. Other students found the course to have had more utility, and offered insightful comments, such as, "There is more to specs than appears at first glance", "I believe that the most important aspect is specs…Before I would not have even really considered this…let alone made it a top priority", and "view before [was]…as long as the stinking code works everything is fine."

Table 2: Responses to Question 1, End of OSU Sequence

Response	Before	After
Readable and understandable	9	13
Design before coding	2	9
Reusable	0	10
Efficient	4	6
Modular	0	5

Table 3: Responses to Question 1, End of CS2 at WVU

Response	Before	After
Readable and understandable	5	10
Design/specify before coding	4	12
Reusable	1	5
Efficient	1	6
Reasoning and testing	0	7

Table 4: Responses to Question 2, End of OSU Sequence

Response	Before	After
No view, monolithic code with some procedures	24	3
Somewhat component-based	6	0
Component-based	0	24
Other	6	9

Table 5: Responses to Question 2, End of CS2 at WVU

Response	Before	After
No view, monolithic code, some procedures	12	0
Component-based, specification-based	4	15
Other	3	4

Attitudinal Evaluation

Table 6 contains the attitudinal survey along with our pre-survey hypotheses on trends. In the table "+" denotes statements where we expected more students to tend towards agreement at the end of the sequence compared to the beginning, and "–" denotes statements for which we expect more students to tend to disagree at the end. For statement #19, we agreed early-on that the wording was ambiguous and could not agree on which trend to predict. Statements 23 to 29 were administered only at OSU.

For each statement, students were asked to make one of six choices: Strongly disagree, disagree, moderately disagree, moderately agree, agree, and strongly agree. Intentionally, the students were forced make an agree/disagree decision without providing them a way out, such as "other". At OSU, students took four surveys: one at the beginning of each of the three courses and one at the end of the last course. At WVU, the students also did four surveys, but one each at the beginning and at the end of the two

semester courses. (While most students at OSU go through the sequence in more or less consecutive quarters, at WVU students tend to take other courses between the second semester CS course and the junior-level software engineering course.)

Table 6: Attitudinal Survey Questions/Expected Trends

1. Software development is a challenging activity. (+)
2. If I worked for a company and was asked to develop 10,000 lines of software to solve a problem, I feel capable of designing and developing that software. (+)
3. The difficulty in understanding and modifying a 10,000 line software system has more to do with the style in which the software is written, and less to do with how smart I am. (+)
4. The difficulty in understanding and modifying a 10,000 line software system has more to do with the style in which the software is written, and less to do with the programming language in which it is written. (+)
5. Software development can benefit from carefully designing each component before coding it, as opposed to quickly coding and experimenting with it. (+)
6. Some of my friends and I know and follow a disciplined approach to software design and development. It is therefore easy for us to write parts of a large software system separately and then put them together easily. (+)
7. The main challenge of software construction lies in coding the design in a programming language, and not so much in specifying what needs to be done. (–)
8. It is possible to show that a software component works without actually running it on the computer. (+)
9. Programming language statements -- even in the absence of comments -- are well-suited for describing precisely what a software component is supposed to do. (–)
10. Successful software development has a lot to do with mathematical models and proofs. (+)
11. To understand what a 10,000 line program does, you need to understand every procedure and function used in that program. (–)
12. When I try to develop correct software, my tendency is to code quickly, but to spend as much time as possible in testing and debugging. (–)
13. It is possible to have an understanding of software that is independent of the programming language used. (+)
14. Correct software can best be constructed by building it from scratch. (–)
15. Software development is so challenging that I often doubt if my programs are 100% correct. (–)
16. My conception of what software is has changed markedly over time. (+)
17. Successful software development is not possible without having a precise mathematical description of what each software component is supposed to do. (+)
18. My conception of how to build software has changed markedly over time. (+)
19. The programming language used is a very important factor in successful software development. (N/A)
20. When working in teams, natural language descriptions of the different components, such as a descriptions in English, are sufficient for communication among team members. (–)
21. Before I run my code on a computer, I make it a practice to hand trace through the statements on example inputs to see if it works. (+)
22. There is really not much difference between what a software part does and how it does it. (–)
Remaining Questions Administered Only at OSU
23. Thinking first about each software component in terms of its implementation details is important for successful software development. (–)
24. Software development can benefit from carefully thinking and reasoning about each software component's correct behavior even before running it, as opposed to quickly testing and debugging it on the computer. (+)
25. Physical metaphors, such as plastic cups and Lego blocks, can help in understanding software. (+)
26. My respect for the challenge of software development has changed markedly over time. (+)
27. It is sufficient to have a natural language description, such as a description in English, of what each software component is supposed to do. (–)
28. It is important to take a consistent approach to design and development of software. (+)
29. How I personally develop software has changed markedly over time. (+)

The last six digits of student identification numbers were used to identify each student. This was essential to track student attitudes, without disclosing the identity of the students. In the analysis given in the paper, we compared the "average" attitudes of students with respect to each question at the beginning and at the end of course

sequence. To avoid noise, we restricted comparisons only to the attitudes of the students who filled out both a "pre-sequence" survey and a "post-sequence" survey. Since the surveys were strictly voluntary, we could identify only 47 students who took both the first and last surveys at OSU between 1997 and 1999, though the total number of students completing surveys is much larger (about 700 at the beginning and about 150 at the end). Another problem that made it difficult to track individuals was just that several students failed to include their identifications on one or more of the surveys. However, the sample size of 47 at OSU is a substantially large enough to allow initial conclusions, though further studies are needed. For similar reasons, only 15 surveys were matched using student identifications at WVU. Also at WVU, only some of the sections used the CBSE approach in the fourth semester software engineering course (i.e., the second course in the sequence).

Tables 7 and 8 present a summary of results from attitudinal evaluations at OSU and WVU, respectively. (A large number of students did not respond to question #29 at OSU, hence it is not included in the analysis.)

Table 7: Summary of Attitude Changes at OSU

No.	Before	After	Diff	P-value	Significant?
1	1.8	4.7	+2.9	< 0.01	High
2	3.5	4.1	+0.6	0.05	Marginal
3	2.1	4.5	+2.4	< 0.01	High
4	2.5	4.5	+2.0	< 0.01	High
5	2.0	4.9	+2.9	< 0.01	High
6	3.1	4.1	+1.0	< 0.01	High
7	3.9	3.0	-0.9	< 0.01	High
8	3.1	4.5	+1.4	< 0.01	High
9	3.7	3.4	-0.3	0.8	No
10	2.9	4.0	+1.1	< 0.01	High
11	3.0	3.2	+0.2	0.3	No
12	3.5	3.2	-0.3	0.8	No
13	2.2	4.7	+2.5	< 0.01	High
14	3.6	3.4	-0.2	0.4	No
15	3.5	3.4	-0.1	0.6	No
16	2.9	4.6	+1.7	< 0.01	High
17	2.9	4.3	+1.4	< 0.01	High
18	3.0	3.7	+0.7	< 0.01	High
19	2.4	4.9	+2.5	N/A	N/A
20	3.3	3.4	+0.1	0.7	No
21	3.5	2.6	-0.9	0.99	Neg. High
22	2.7	4.9	+2.2	0.99	Neg. High
23	2.6	3.6	+1.0	0.99	Neg. High
24	2.0	4.0	+2.0	< 0.01	High
25	2.4	4.8	+2.4	< 0.01	High
26	2.4	4.5	+2.1	< 0.01	High
27	2.6	5.0	+2.4	0.99	Neg. High
28	2.3	4.8	+2.5	< 0.01	High

Table 8: Summary of Attitude Changes at WVU

No.	Before	After	Diff	P-value	Significant?
1	5.2	5.0	-0.2	0.6	No
2	3.8	4.6	+0.8	0.09	Marginal
3	4.3	4.8	+0.5	0.6	No
4	4.3	5.1	+0.8	0.02	Yes
5	5.6	5.6	0.0	0.7	No
6	4.5	4.8	+0.3	0.3	No
7	3.2	1.8	-1.4	< 0.01	High
8	3.3	5.2	+1.9	< 0.01	High
9	3.6	2.7	-0.9	0.07	Yes
10	4.1	4.5	+0.4	0.2	No
11	3.7	2.2	-1.5	< 0.01	High
12	3.2	2.5	-0.7	0.07	Marginal
13	4.7	5.7	+1.0	< 0.01	High
14	3.9	2.1	-1.8	< 0.01	High
15	4.0	3.5	-0.5	0.1	No
16	3.6	4.9	+1.3	0.02	Yes
17	3.1	4.5	+1.4	< 0.01	High
18	4.5	5.3	+0.8	0.02	Yes
19	3.5	2.4	-1.1	N/A	N/A
20	4.2	2.8	-1.4	0.02	Yes
21	3.9	4.4	+0.5	0.4	No
22	3.3	2.3	-1.0	0.02	Yes

It is important to note that the surveys come from students who took the courses at different times under different instructors (including teaching assistants at OSU), and therefore, it is reasonable to conclude that the impact of specific instructors is minimal.

In each table, the first column indicates the statement number. Each response was given a score of 1 to 6, ranging from 1 for "strongly disagree" and 6 for "strongly agree". The second and third columns show the average agreement scores for all students for each statement at the

beginning and at the end of our course sequences, rounded to the nearest tenth. An average score of 3.0, for example, denotes that the average response was "moderately disagree". The fourth column indicates the change in average response before and after the sequence. Column 5 contains P-values derived from one-sided paired t-tests on the raw data. This involves computation of the change in the attitude of each student using their identification numbers, and it determines whether the average change in the previous column is statistically significant. P-values below 0.01 denote high significance, and those between 0.01 and 0.05 denote significance. P-values that are above 0.99 denote high significance in the wrong direction. Strictly, such P-values suggest that 2-sided evaluations should have been used.

The OSU survey summary shows that the attitudes changed *highly significantly* for 16 out of 27 statements, towards our expectations. No trends were seen for 6 statements, and 4 statements showed significant trends that were against our expectations. For one statement, the change was marginal. In the WVU survey, no trends were against our initial expectations. For 12 out of 22 statements, the attitude changes are significantly or highly significantly towards our expectations. For 2 statements, changes are marginally significant. For the other 7 statements, no appreciable changes could be detected. These results strongly suggest that the trends are not coincidental.

For several statements in the OSU survey, the average attitude score changes are highly significant, with average changes around 2.0. For example, in the case of statement #3, students shifted their attitudes toward believing that with regard to understanding and maintaining a system, their intelligence is a less important factor than how the software is built. We suspect that changes in such attitudes, as in the notion that software can be understood independently of the particular programming language involved (#4) and the importance of designing before coding (#5), would be difficult to duplicate without considerable impact from the educational philosophy and approach. At OSU, students follow a rigorous discipline of software construction using a single language C++ in their three-course sequence. The impact of this approach is seen in their attitudes on statements such as #6, #16, #18, and #28 concerning discipline and conceptions of software. Students also found that physical metaphors, such as plastic cups and Lego blocks, can help considerably in understanding software (#25).

In the case of some statements, e.g., #9, it is unlikely that students would have been able even to understand such a subtle statement at the beginning of the sequence. Nonetheless, most students agreed at the beginning with this one, so their attitudes did not change much as a result of taking the courses.

The trends in responses to some questions indicate that the uncertainty in student answers (around 3.5 average) was not altered by the courses. For example, students at OSU remained ambivalent about the broad role of precise specifications (#20), but felt required to assert the rightful role of natural languages in software engineering (#27), despite the great emphasis the courses placed on formalization. For these questions, the attitudes in the WVU survey show change in the anticipated direction. This difference is probably explained by the fact that the WVU course sequence includes an explicit requirements analysis part using natural language descriptions. Taught this explicit connection between natural and formal specification languages, students were likely able to see the importance of both in large-scale software engineering. Students in the OSU survey apparently equated the use of formal specifications in those courses to mean avoiding natural language descriptions completely, despite repeated examples showing that formalization in no way implies that natural language descriptions are taboo.

The WVU survey summary contains trends towards our goals. The attitude changes are large for statements #7, #8, #13, #16, #17, and #20. Like the OSU students, WVU students emphasize the importance of specifying before coding (#7) and agree with the notion that it is possible to have an understanding of software independent of the language used (#13). Both OSU and WVU students agree that the courses have changed their conceptions of software (#16). WVU students emphasize the role and importance of specifications more than their OSU counterparts. As seen from the trends for statements #17 and #20, WVU students are considerably more in agreement on the importance of precise specifications. The trend for statement #19 (along with #13) also shows that the students view programming language to be less of a factor in software engineering. These positive trends at WVU may be a result of the use of RESOLVE notation, unencumbered by C++, in some classes.

For some questions, there is little change in WVU student attitudes compared to OSU students. This is the case for statements such as #1, #3, #4, and #5. This is possibly due to a combination of factors. At OSU, students entered the sequence beginning with their first course in CS except for having had a basic introductory programming course (perhaps in high school), and almost all change in their attitudes towards software came during the sequence. At WVU, students had already had a first course in CS, and in some cases and on some statements their attitudes were already where we wanted them to be, leaving little room for significant improvement.

Lessons Learned

This section presents a few of the difficulties encountered in teaching our CBSE approach in multiple courses, and evaluating its impact. For the principles to be taught effectively, it is important to have multiple faculty members who understand and agree with the underlying philosophy, and who are willing to communicate the principles enthusiastically in their courses. We were fortunate in this regard. It is also essential to develop and continuously revise course materials based on student feedback using qualitative evaluations.

Evaluation of educational results, given the variety of confounding factors, is non-trivial. It is a long-term activity involving many people, and several things can go wrong. For example, it is important to have the right words and context in attitudinal survey questions and statements. It is a challenge to develop unbiased wordings which are understandable to students before taking the courses and which can produce meaningful information about attitude changes. In some cases, our wordings were such that the responses tended to be where we wanted them to be right at

the beginning, leaving little room for improvement. In spite of our best efforts, we cannot be sure that the initial responses from students were based on reasonable interpretations of the words. We also realized too late that the wording of some questions was not as balanced as it should have been, resulting in the desired trends being toward agreement for more than half the statements (including the first six). Fortunately, this seems to have had little impact on student responses, because there is considerable disagreement at the beginning of the course. Finally, it is important to have effective means to get most students to answer the surveys and to include their identifications without fail for evaluation purposes.

It is clear that essay-type questions used at the end of the courses provided useful information. When employed in conjunction with other surveys, they can be powerful tools in helping justify/reject initial hypotheses. However, qualitative evaluations need to be analyzed manually, because automated key word analysis is suspect. Such evaluations should be used to suggest future quantitative instruments.

5 CONCLUSIONS

It is becoming increasingly clear that academic institutions should educate undergraduate students in advanced component-based software engineering principles. It is also clear that such education should start early in the curriculum. When software engineering is taught as a set of add-on principles near the end of the degree, students find it hard to "unlearn" previous habits and do not have sufficient time to experiment with software engineering principles in their education. To address these problems, we have adapted a formal approach to CBSE for introduction in introductory undergraduate CS courses, without displacing classical concepts normally taught in those courses. By trying and evaluating this approach at two major public institutions, we have illustrated that other schools might be able to adapt and apply similar principles in their settings.

An excellent collection of software engineering education efforts, including education in formal methods principles at the undergraduate level, may be found in [9]. Our work differs from earlier studies not merely in technical CBSE details, but also in our methods of evaluating the impact. Our study has shown that students are able to appreciate the significance and understand principles of formal specifications and component-based reuse. We have employed qualitative and quantitative evaluations to study the impact of our education on student learning and attitudes over a 3-year period. The summary results are statistically significant, showing intended outcomes on most indicators. While we have studied the impact immediately after the courses, additional studies are needed to determine if the impact on attitude changes on students persists in the long-term.

ACKNOWLEDGEMENTS

Many people contributed important ideas to this work and/or made helpful comments about drafts of this article. We would especially like to thank Sheila Arbaugh, Paolo Bucci, Greg Kulczycki, Mike Henry, and Bill Ogden.

We also gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-9311702, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office, and from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, the U.S. Department of Defense, or Microsoft.

REFERENCES

1. "Special Feature: Component-Based Software Using RESOLVE," *ACM SIGSOFT Software Engineering Notes 19*, No. 4, Eds. M. Sitaraman and B. W. Weide, October 1994, 21-67.
2. Ford, G, *SEI Report on Undergraduate Software Engineering Education*, CMU/SEI-90-TR-003 ADA223881, 1990.
3. Parnas, D.L., "Teaching Programming as Engineering", in ZUM `95: The Z Formal Specification Notation, *9th International Conference of Z Users*, Bowen J.P., Hinchey M.G. (eds.), *Lecture Notes in Computer Science 967*, Springer-Verlag, 1995, pp. 471-481.
4. Tomer, T.S., Baldwin, D., and Fox, C. J., "Integration of Mathematical Topics in CS1 and CS2," *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, ACM, 1998, 364-365.
5. Weide, B.W., *Software Component Engineering*, OSU Reprographics, Columbus, OH, 1996.
6. Sitaraman, M., *An Introduction to Software Engineering Using Properly Conceptualized Objects*, WVU Publications, Morgantown, WV, 1997.
7. Long, T.J., Weide, B. W., Bucci, P., Gibson, D. S., Hollingsworth, J., Sitaraman, M., and Edwards, S., "Providing Intellectual Focus to CS1/CS2," *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, ACM, 1998, 252-256.
8. Long, T.J., Weide, B. W., Bucci, P., and Sitaraman, M., "Client-View First: An Exodus from Implementation-Biased Teaching", *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, ACM , 1999, 136-140.
9. Software Engineering Education, Eds. Coulter, N. S., Gibbs, N. E., *Annals of Software Engineering 6*, 1998 (1-90, 365-453).
10. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software-Component Behavior," in Frakes, W.B., ed., *Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse),* Springer-Verlag LNCS 1844, 2000, 266-283.
11. Sitaraman, M., Weide, B. W., Long, T.J., Ogden, W. F., "A Data Abstraction Alternative to Data Structure/Algorithm Modularization," *LNCS 1766*

Volume on Generic Programming, Eds. D. Musser and M. Jazayeri, Springer-Verlag, 2000, 102-113.

12. Edwards, S., Shakir, G., Sitaraman, M., Weide, B. W., and Hollingsworth, J., "A Framework for Detecting Interface Violations in Component-Based Software,"*Proceedings of the Fifth International Conference on Software Reuse*, IEEE Computer Society Press, Victoria, Canada, June 1998, pp. 46-55.

Appendix: Example Component Specification

This appendix shows the specification of a *List* type in RESOLVE notation (reproduced from [10]). *List_Template* is a generic **concept** (specification template) which is parameterized by the type of entries to be contained in lists. To provide abstract mathematical explanations of the operations, an object of type *List* **is modeled by** an *ordered pair of mathematical strings* of entries. A string is similar to, but simpler than, a "sequence" because it does not explicitly include the notion of a position. The operator "*" denotes string concatenation; "<*x*>" denotes the string containing the entry *x*; and "|*s*|" denotes the length of *s*.

Conceptualizing a *List* object as a pair of strings makes it easy to explain insertion and removal from the "middle". A sample value of a *List* of *Integer*s object, for example, is the ordered pair (<3,4,5>,<4,1>). Insertions and removals can be explained as taking place between the two strings, e.g., at the left end of the right string.

The declaration of type *List* introduces the mathematical model and says that an object of type *List* initially (i.e., upon declaration) is "empty": both its left and right strings are empty strings. Each operation is specified by a **requires** clause (precondition), which is an obligation for the caller; and an **ensures** clause (postcondition), which is a guarantee from a correct implementation. In the postcondition of *Insert*, for example, *#s* and *#x* denote the incoming values of *s* and *x*, respectively, and *s* and x denote the outgoing values. *Insert* has no requirement, and it ensures that the incoming value of *x* is concatenated onto the left end of the right string of the incoming value of *s*; the left string is not affected. Notice that the postcondition describes how the operation **updates** the value of *s*, but the return value of *x* (which has the mode **alters**) remains unspecified.

Given this specification, students act as clients and use lists in problem solving within the first few weeks of their second quarter/second semester course. They use a specification-based "natural" or forward reasoning method to reason about correctness [10]. Only later they learn how to implement lists using pointer structures. In addition to classical examples such as Lists, students also see data abstractions that result from recasting classical algorithms as objects [11], and aspects of specification-based interface violation testing using wrapper components [12].

RESOLVE specifications use a combination of standard mathematical models such as integers, sets, functions, and relations, in addition to tuples and strings. The explicit introduction of mathematical models allows use of standard notations associated with those models in explaining the operations. Our experience is that this notation—which is precise and formal—is nonetheless fairly easy to learn to understand even for beginning computer science students, because they have seen most of it before in high school and earlier.

```
Concept List_Template (type Entry)
  Type List is modeled by
        (left: string of Entry,
         right: string of Entry)
    exemplar s
    initialization ensures
      |s.left| = 0 and |s.right| = 0

    Operation Insert (
        alters x: Entry
        updates s: List
      )
     ensures s.left = #s.left and
          s.right = <#x> * #s.right

    Operation Remove (
        replaces x: Entry
        updates s: List
      )
     requires  |s.right| > 0
     ensures s.left = #s.left and
          #s.right = <x> * s.right

    Operation Advance (
        updates s: List
      )
     requires |s.right| > 0
     ensures s.left * s.right =
              #s.left * #s.right and
          |s.left| = |#s.left| + 1

    Operation Reset (
        updates s: List
      )
     ensures |s.left| = 0 and
          s.right = #s.left * #s.right

    Operation Advance_To_End (
        updates s: List
      )
     ensures |s.right| = 0 and
           s.left = #s.left * #s.right

    Operation Left_Length (
        restores s: List
      ): Integer
     ensures Left_Length = |s.left|

    Operation Right_Length (
        restores s: List
      ): Integer
     ensures Right_Length = |s.right|

end List_Template
```