Developing a Learning Progression that Integrates Communication in an Undergraduate CS/SE Curriculum

Michael Carter, Robert Fornaro, Sarah Heckman, Margaret Heil

North Carolina State University

May 25, 2012

Abstract

There are global concerns that communication (writing, speaking and teaming) skills of recent computer science/software engineering (CS/SE) graduates need improvement. One reason for poor communication skills is that instruction in communication is typically removed from the CS/SE curriculum, farmed out to technical communication courses focusing on general skills rather than those specific to CS/SE. However, CS/SE faculty are the experts in the specific kinds of communication CS/SE graduates need to succeed. Preliminary results of an NSF-funded project to design learning outcomes and teaching practices that will enable CS/SE faculty to integrate communication throughout their curricula are presented. The approach is one of building a learning progression in communication across a given curriculum. The proposed learning progression is based on two concepts. One is genre theory, which highlights patterns of communication founded on certain often-repeated situations and the typical responses to those situations. CS/SE is characterized by many of these recurring types, or genres, of communication, such as problem definitions, expression of system requirements and design, test plans/results, code and comments. Thus, genre theory provides a way of identifying and defining the recurring types of CS/SE-specific communications that are likely already included in a CS/SE curriculum. The other concept is the growth in communication skills across a CS/SE curriculum. Such growth can be understood as the increasing sophistication of communication abilities in response to the increasing complexity of communication situations created for students as they advance in the curriculum. There are five dimensions of complexity: more complex situations require responses that (1) are longer, (2) are more complicated, (3) require multiple genres, (4) require a greater degree of independence of students, and (5) require more students and more elaborate forms of interaction among them. These two concepts suggest an approach that can be applied to create a framework for communication across the CS/SE curriculum: (1) identify the communication goals that students are expected to achieve in the curriculum, typically the goals represented in the senior capstone course; (2) analyze the expectations in the capstone according to communication genres and levels of complexity; (3) begin with the first course students are expected to take and, for this and each successive course, map out an increasing complexity of situations for each genre in the capstone. The discussion begins with an analysis of a typical capstone course and then moves to analyses of computer science (level 1), computer science (level 2), and software engineering courses. An example of a learning progression for communication skills, including communication genres and levels of complexity that are appropriate for each course, is described.

1. INTRODUCTION

The gap between the communication abilities of recent graduates and the expectations of managers in Computer Science and Software Engineering (CS/SE) and other engineering fields is well documented [1], [6], [8], [13], [14], [15], [16], [17]. One of the reasons this gap exists is that most instruction in engineering communication occurs outside engineering departments, typically in technical writing and oral communication courses. And even when communication is taught in engineering departments, it is typically done so by instructors trained in English or oral communication, not in engineering [14]. Although these instructors may be experts in the general types of communication of the engineering workplace, they are usually not familiar with the field-specific types of engineering workplace communications [14], [18].

One solution to this problem is to integrate communication within the engineering curriculum. One of the most influential theories of learning to emerge over the last twenty-five years is based on research by Lave and

Wenger into apprenticeships, called *Legitimate Peripheral Participation* [9]. This research suggests that learning is most effective when it takes place in situations that are the same or similar to those in which learners will apply what they have learned, in other words, situations like apprenticeships. The key is that novices learn by doing what experts in the field do and learn through the guidance of experts, though without the high expectations and full responsibility of the experts. This theory suggests that people learn by engaging in the legitimate activities of the field in an environment in which those activities are normally done. The novices may begin on the periphery of the field as they are learning, but they are still considered apprentice members of the field as they move toward the full membership of experts.

We suspect that most CS/SE faculty are more comfortable with the apprenticeship model of learning in regard to technical skills than communication. This may be because they do not consider themselves as experts in communication. However, because CS/SE faculty are trained in the discipline and often have industry experience, they have a far greater expertise in the types of communication used by professionals in the field than instructors in English and oral communication [5], [18]. In terms of the apprenticeship model, technical skills and communication skills are on an equal footing. If CS/SE students are to become capable as professionals in their field, they need to receive instruction and experience in communication in the field. Therefore, communication should be integrated into technical education to the same extent as the two are integrated in professional environments in CS/SE.

One major hurdle in achieving this goal is *how* to integrate communication into the undergraduate CS/SE curriculum. This technical report will provide a process for determining the progression of communication outcomes into the curriculum. We will offer strategies for identifying the types, or genres, of communication that are specific to the field and integrating these types of communication in a way that promotes growth in abilities throughout a curriculum. We will provide an example of the application of these strategies to a CS/SE curriculum on several common genres from the software development lifecycle, specifically software requirement specifications, design expressions, code and code comments, and test expressions.

2. Genre in Computer Science and Software Engineering

To be able to provide coherent instruction and communication experiences for students in CS/SE, we need to identify the specific genres of communication that define the field. In a classic definition by Miller [10], genres are "typified rhetorical actions based on recurrent situations." In other words, certain categories of communication develop in response to situations that recur with some frequency. This concept may sound familiar to computer scientists because it is similar to design patterns associated with object-oriented software [7]. Both are reusable forms or templates that can be applied to commonly recurring situations.

CS/SE is characterized by many recurring situations that have led to a set of typical responses, or forms of communication, to those situations. In one example of such a situation, a client needs to communicate a problem that can be solved with software to a team of software engineers who may not be familiar with the client or the client's problem. A communicator may serve as liaison between the client and the software engineers. The communicator must respond by providing the information necessary for the software engineer to successfully solve the client's problem. The form of communication that has developed to address this situation is software requirement specifications (SRS), which describes in user terms what a program should. Over time, the SRS has come to be defined by a set of conventions that the communicator can call upon to achieve the particular purpose for the specific audience in this recurring situation. For example, the SRS can be provided in terms of functional or non-functional requirements, use cases, or user stories, etc. As another example, the expression of a design may be understood as a typical response to the recurring situation in which functions necessary for the software to perform in order to solve the given problem are explained in a way that provides a blueprint for implementing code.

Though it is not possible to account for every recurring situation and communication response in the CS/SE field, there are some that are classic. The recurring situation and communication response, or genre, for these classic genres are listed in Table 1. As the list of genres in Table 1 suggests, communication that is specific to CS/SE

plays a critical role in the apprenticeship of students in the field. It introduces them not only to the individual situations and responses that define the field but also to the broader software development life cycle consisting of these genres. The responses are a part of the legitimate participation of people in CS/SE, integrated with the technical production they are asked to do. Each type of communication is a response to frequently recurring situations in the field. All CS/SE genres incorporate purposes and audiences that help students to understand why they are being asked to engage in the communication. In contrast to Orr [11], we have focused on genres specific to CS/SE, not including more general professional genres such as progress reports, trip reports, and research reports. We also make a distinction between genre and medium. Whereas genre is the recurring CS/SE situations and communication responses to those situations, medium is the *means* by which we communicate, such as email, phone calls, PowerPoint presentations, and whiteboards.

Table 1. Common genres in SE as identified by the recurring situation and communication response

Recurring Situation	Communication Response
Client has a problem requiring a software solution	Definition of a SE problem
Client wants to communicate software needs with a team of software engineering unfamiliar with the client or client's problem	SRS
Software engineers need to identify functions necessary for the software to solve a problem	Design expression
Software engineers need to implement a design that is readable and maintainable	Code and comments in the code
Software engineers need to help new team members get started with development	Developer guide
Software engineers need to determine if the code is right and if the code does the right thing	Test plan
Software engineers need to allow for automated testing	Testing
Testers need to identify incorrect functionality in the product	Bug report
Software engineers need to help the client with installing and maintaining their system	Installation and maintenance guide
Software engineers need to help the client use their system	User guide

3. Growth in Communication Throughout the CS/SE Curriculum

The concept of Legitimate Peripheral Participation provides a model for growth in student learning that could be used to shape a curriculum that integrates communication. Students enter our programs as novices in CS/SE, on the periphery of the field. The goal is to move them toward full membership as professionals in the field. In most CS/SE curricula, the capstone course is a bridge between the classroom and the workplace [12]. In most of these courses, students are placed in situations that mimic those they will encounter as professionals. The curriculum, then, should be designed to develop students as professionals before the capstone, with the technical and communication skills they need to succeed.

There are two principles that guide this development of students as capable communicators in CS/SE [5]. One is that students should participate in the legitimate communication practices of the field throughout the curriculum, from first through senior years. Implicit in this principle is that there is no inherent hierarchy of genres that would lead to a stepped progression from one to the other. Development in communication is not a matter of first mastering one genre and then another in stepwise fashion through the list of CS/SE genres. Rather, development depends on students having ample opportunities to communicate in a broad range of disciplinary genres throughout their programs.

The second principle is that progressive development as communicators in a field is a function of the increasing sophistication of student responses to the increasingly complex situations of assignments. The idea here is that the students' communication becomes more sophisticated as the situations they are responding to become more

complex. They may be asked to write a simple, perhaps even one-sentence, SRS in an introductory computer science course and develop progressively in writing more and more sophisticated specifications culminating in writing a full specification for an external client in the capstone course.

But the question is what does that growth in sophistication look like? How do CS/SE faculty create progressively complex situations throughout a curriculum? The first principle suggests that we should incorporate all genres at all levels of the undergraduate curriculum. The second principle suggests that the situations and responses increase in sophistication as students move through the curriculum. We have integrated these principles into five dimensions of situational complexity that can guide development of a learning progression that can be used to promote growth in sophistication of students' communication. We will illustrate these dimensions with common genres in CS/SE [5].

The five dimensions of situational complexity are that:

1. *More complex situations require longer responses*. The goal is not length simply for the sake of length. Instead, it is to increase the extent of the engagement of students in a project.
2. *More complex situations require more complicated responses*. By "more complicated," we refer to the literal meaning of the word as something having increasingly more folds or plies.
3. *More complex situations require responses in multiple genres*. An assignment becomes more complex as students build a stronger sense of how a given genre fits together in the broader CS/SE enterprise.
4. *More complex situations require a greater degree of independence on the part of students*. Students work with less dependence on the instructor and more dependence on themselves.
5. *More complex situations require more students to contribute to the completion of a project*. Increasing the number of students on teams, different venues for students interaction (from in-class to a combination of in-class and out-of-class to all out-of-class), the level of formality of the teamwork (from informal groups to increasingly formal ones with set roles for students and records that must be maintained), and the amount of time the team is expected to work on the project (from a few minutes in class to a semester- or year-long project).

4. Creating a Learning Progression for a CS/SE Curriculum

As we have suggested, Lave and Wenger's [9] model of apprenticeship offers a way to conceive the development of CS/SE students as they move from novice toward expertise in the field. Legitimate Peripheral Participation is critical to enabling students to achieve that expertise. Students learn by doing what experts in the field do, which includes communication. We suggest that an effective CS/SE curriculum should be shaped so that it actively engages students in the development of both technical and communication abilities. In this section, we will describe a method for shaping a curriculum that integrates communication in a way that advances both the communication and technical abilities of students. The method consists of two stages, first identifying program outcomes for technical and communication abilities and second shaping a curriculum for developing students as effective communicators in the field.

Identifying program learning outcomes is important because planning on the curricular level requires that faculty understand what students should be able to do by the time they graduate. The idea is that a student who graduates with a CS/SE degree may be distinguished in his or her preparation from a person who is simply good at programming. A key marker is that the CS/SE graduate brings particular *ways of thinking* about CS/SE that the other person typically does not possess. A faculty group at NC State set out to define those ways of thinking and related learning outcomes are listed in Table 2 (the learning outcomes related to the science of computing are not included here so as to focus on software development).

How do students develop these ways of thinking? Communication plays an essential role because the different genres of CS/SE shape particular ways of thinking, both individual genres and collective genres that define the software development lifecycle. For example, the SRS defines a particular way of thinking about what a program is supposed to do, a way of thinking that both writer and audience are trained to expect. But the SRS is also an element in a broader set of genres that defines a broader way of thinking in CS/SE: defining a problem,

Table 2. Selected program learning outcomes defining the ways of thinking of software development

To demonstrate that graduates can reason effectively about computing and develop software, they should be able to:
1. Recognize and define a problem related to a specific scenario that can be solved with a software application. Describe how the end-users or internal actors within a system intend to use the application to be developed. Gather and analyze information that allows for requirements that will solve the problem to be created, validated, verified and if necessary, revised.
2. Create and express a design for an underlying abstract model of computation that accommodates defined system requirements—including considerations of privacy, security, and efficiency—so that a developer can implement the application. Review the design to ensure it can accomplish the requirements and, where it does not, redesign until it meets the requirements.
3. Implement software conforming to a specified design so that it is usable, testable and modifiable by others. Review the implementation to ensure it meets the system requirements and conforms to design and, where it does not, correct the implementation until it meets the requirements and design.
4. Plan and execute appropriate tests in order to identify ways in which the software does not meet the requirements and, where it does not, to redesign, implement and retest until it meets the requirements.

determining requirements, creating a design, etc. To a large extent, this way of thinking delineates what it means to be a professional in the field. And development as a professional means mastering the genres that define that way of thinking.

Thus, the next step after describing the way of thinking in CS/SE is to specify the communication abilities that go hand in hand with the technical ways of thinking. An example created by the NC State faculty group is listed in Table 3. Students both learn and demonstrate that they have learned the technical ways of thinking by engaging in these forms of communication. For example, students are expected to "Recognize and define a problem related to a specific scenario that can be solved with a software application. Describe how the end-users or internal actors within a system intend to use the application to be developed." Students both develop and demonstrate that ability by "Present[ing] in writing or orally a critical assessment of a problem situation defined by a need for software to be developed for solving the problem: (a) collect information from sponsors, end-users, and on-site observations, (b) analyze that information (c) use the analysis to define the problem in terms of the stakeholders' needs and goals for addressing those needs." Students learn the technical way of thinking associated with defining a problem to be solved by software by formally defining such problems. Thus, the program outcomes produced in this way present the technical ways of thinking in CS/SE and the ways of communicating that enable students to master those ways of thinking.

The next stage is the analysis and revision of a curriculum by which students develop the expected ways of thinking and communicating. This process may be done in five steps.

1. Align the capstone course with the program outcomes. As we have noted, the typical role of the capstone is to provide a bridge for students between the university and the workplace. It is where students engage in activities that encourage them to synthesize and apply what they have learned in a context that mimics the workplace. Thus, ideally, a capstone would reflect the ways of thinking and communicating in the program outcomes. So addressing the curriculum would begin with a review of the capstone in light of the outcomes and, where it is found wanting, to revise it.

2. Analyze the capstone for complexity of its communication assignments. If the capstone represents the ultimate student experience in preparation for the workplace, then it also serves as a target for the preparation of students earlier in the curriculum. A curriculum should be structured so that it enables students to develop the technical and communication abilities necessary to take full advantage of the opportunities in the capstone. We have defined development of communication (and thus technical) abilities as a growth in the sophistication of the communication, a growth that is guided by the increasing complexity of communication assignments.

Table 3. Selected communication outcomes focusing on software development for students to learn and to demonstrate that they have learned the ways of thinking in Table 2

To demonstrate that graduates have achieved the general program learning outcomes, they should be able to:
1. Present in writing or orally a critical assessment of a problem situation defined by a need for software to be developed for solving the problem: (a) collect information from sponsors, end-users, and on-site observations, (b) analyze that information (c) use the analysis to define the problem in terms of the stakeholders' needs and goals for addressing those needs.
2. Write requirements representing the stakeholders' needs and goals in such a way that the requirements can be applied in a design by others.
3. Read requirements for various purposes, such as to inspect and correct them, to validate them as meeting the user's needs, to revise them so that they better meet user's needs, to implement them in a design, and to identify what students don't know and what they need to know to create code.
4. Write a design that accommodates the defined system requirements—including considerations of privacy, security, and efficiency—so that a developer can implement the application.
5. Read a design for various purposes, such as to ensure it can accomplish the requirements and, where it does not, redesign until it meets the requirements and to translate it into code.
6. Write a program to conform to a specified design so that it is usable, testable, and modifiable by others.
7. Write a narrative description of code, including a list of file names or directories included.
8. Read code and comments for various purposes, to find and correct errors in syntax and semantics, to determine what a program is supposed to do, to revise a program so that it accomplishes what it is supposed to do, to modify a program for different purposes, to ensure that a program conforms to system requirements and conforms to design, to provide productive feedback to those who created it, to continue a program begun by someone else, and to apply it to new uses.
9. Write a developer guide that is appropriate to the audience.
10. Write a user guide that is appropriate to the audience.
11. Present in writing or orally a test plan and results of testing that identifies ways in which the software does not meet the requirements.
12. Work effectively in teams: (a) develop ground rules to guide the team's approach to work; (b) define roles so that expectations of team members are clear and followed, (c) create agendas and minutes for team meetings; (d) interact with other team members in ways that assure the productive contributions of all team members; (e) create specific action items for each member and then hold him or her accountable; (f) identify, create, and manage the tools that enable teams to work effectively; (g) resolve conflicts among team members.

Therefore, as a goal for the development of students in the curriculum, the capstone's assignments need to be analyzed for their complexity in the five dimensions we have presented.

3. Assess the present curriculum for preparation of students for the capstone. This assessment is necessary for revising the curriculum. The question to be answered is, where in the curriculum are students engaging and receiving instruction and guidance in the genres they must use in the capstone? That information could be derived from a survey of faculty and/or students or an analysis of syllabi. The first principle we discussed above suggests that students should be given opportunities to participate in the genres of the field at all levels. The goal of this assessment, then, is to identify gaps in students' preparation, what genres are missing or are being under taught. For practical purposes, it may be better to focus only on the required core courses in the curriculum.

4. Revise the curriculum so that it better prepares students for the capstone. Any gaps for genres in the curriculum exposed by the assessment are opportunities for improvement. Where there are genres missing or used in only one course, where else should they be used? The goal is not just to determine where genres are to be included but also the growth in sophistication of students' work in those genres. It is important that the appropriate levels of complexity of assignments be considered. For example, students may experience a genre for the first time in an introductory course by reading it rather than writing it. In that instance, instruction would be designed to make students aware of the name of the genre, its role in the development of software, and

perhaps its features. In this way, many genres may be introduced to students in their first required course to initiate their development in the field.

5. Assess the revisions to the curriculum to determine if students are better prepared when they reach the capstone. Did the addition or modification of how genres are presented in earlier classes increase the ability of students to perform in the capstone course?

5. Creating a Learning Progression: An Example

In this section, we will describe our experience at NC State in creating a learning progression for the development of technical and communication abilities in our curriculum. We follow the steps outlined in Section 4, and provide details about our understanding at each step.

5.1 Step 1: Align the Capstone

For step 1, we determined the alignment of our *capstone* course to the program outcomes (as shown in Tables 2 and 3). The capstone course consists of a semester long team project. Student teams, ranging from a size of two to four, complete a project for an external customer. The teams follow the software development lifecycle by eliciting requirements from a customer, creating an SRS, designing, implementing, testing, and delivering a solution that meets the requirements to the customer at the end of the semester. Throughout the capstone experience, the student teams report on their progress using all of the genres listed in Table 1. The capstone focuses primarily on software development, and *we found that it encompasses all the ways of thinking and communicating associated with that area*. Most students do a standard software development project in the course.

5.2 Step 2: Complexity Analysis of Capstone

For step 2, we created a table by which we analyzed the five dimensions of situational complexity of the communication genres of the capstone course (see Table 4). We focused on five key genres: problem definition, SRS, design expression, code and comments in the code, and testing. Table 4 shows each major genre involved in the capstone and describes the situational complexity of each of these genres.

In the capstone course, students create full expressions of all genres that are integrated and supportive of other genres. These expressions are further complicated by complex and changing requirements from the customers. Therefore, many of the expressions require modification over the course of the semester so that all expressions represent the current state of development. The instructors review the expressions and provide guidance, but ultimately it is up to the student team to produce the expressions of all genres. The complexity associated with a team project requires that all members of the student team agree on the delivered communication.

Table 4. Sample capstone complexity analysis

Increasing Complexity Requires / Genre	…longer response	… more complicated response	…response in multiple genres	..greater independence of students	…more students to contribute
Problem Definition	Students are given problem statement, but work with the sponsor/mentor for ~1 week to understand intricacies/demo of problem.	Problem is usually related to sponsor environment, oftentimes unfamiliar to student.	Subsequent components of software development/related documents are dependent on proper statement of problem.	After first mentor meeting, instructor no longer directly participates with students on problem definition.	Student team must agree upon problem definition before creating SRS.
SRS	Students spend about 2 weeks creating and revising the requirements; often change & developed iteratively.	The situations are usually very complicated, with multiple functions and an unfamiliar domain that requires extensive research.	The system requirements are included with all the major genres in the resulting technical report.	Students work independently of the instructor, who acts primarily as a project manager.	Teams: 2-4 students. Most team meetings are outside of class & thus require careful scheduling. Agendas & minutes for all meetings are required, as well as for all meetings with the instructor. They work in teams for the whole semester.
Design Expression	Expression of design accommodates SRS and is usually created iteratively; often begins as diagram; evolves to multiple diagrams with accompanying descriptive narrative.	Changes throughout course of development to accommodate need for effectiveness & efficiency.	Design accommodates requirements and serves as roadmap for code development.	Students are often required to create designs independently of instructor; must defend design decisions when presented to instructors.	Design processed & agreed upon by entire team provides vision for all & launching point for individual implementation.
Code and Comments in Code	Implementation of an entire system based on problem, requirements and design is expected; coding & commenting conventions are defined.	Code will be updated as requirements and design are developed iteratively and as testing reveals issues.	Implemented system must accommodate original system requirements.	Code & comments done independently, but reviewed by instructors.	Team participates in code review.
Testing	Comprehensive testing required: Unit tests, acceptance tests & code reviews of application are expected.	Comprehensive testing means that multiple types of testing and testing frameworks are utilized.	Tests must verify and validate correctness of system implementation with system requirements.	Test frameworks used and test code and documents are chosen and developed individually, but reviewed by instructors.	Code is tested by team members other than coder; code reviews are done by team members.

5.3 Step 3: Assess Earlier Coursework

For step 3, we considered two sources of information to help us assess students' preparation in CS/SE genres in earlier coursework in the curriculum: 1) an informal survey given to capstone students on the first day of the course and 2) an evaluation of earlier courses that identify the genres used and the progression of those genres.

The primary purpose of the survey was to gauge the level of students' preparation in communication in the field to better meet their needs. The Appendix lists the survey questions and a short version of the response sheet. We found that the capstone students' experience with the genres of the field took place almost exclusively in Software Engineering, a junior level course and the prerequisite course for the capstone.

We narrowed the courses analyzed to three courses in the core of the undergraduate curriculum: the first programming course (CS1), the second programming course (CS2), and software engineering (SE). The courses, as taught at NC State, are further described below and the genres evaluated in each course are summarized in Table 5.

Table 5. Summary of genres in evaluated courses				
Course / Genre	**CS1**	**CS2**	**SE**	**Capstone**
Problem Definition	Provided by instructors.	Provided by instructors.	Provided by instructors.	Provided by external client
SRS	Provided by instructors	Provided by instructors as use cases.	Provided by instructors as use cases. During the last project (instructor dependent), the students may be required to write one or more use cases after interacting with a "customer".	Written by students and reviewed by instructors.
Design Expression	Provided by instructors.	Students create a design document as part of Iteration 0. Design imposed on the students for later iterations (to facilitate automated grading). At project conclusion, students reflect on their design document vs. the instructor's design document.	Students design their modifications to an existing system.	Written by students and reviewed by instructors
Code and Comments in Code	Students conform to instructor design and comment their code in their own words.	Students conform to instructor's design and comment their code in their own words. Javadoc HTML files are generated from student's in-code commenting	Students implement their design and comment their code.	Written by students and reviewed by student teams in code inspections.
Testing	Students create black box test plans and implement white box test code	Students create and revise black box test plan. Students show actual results of running the tests. Students write automated unit test code and achieve 100% method coverage on all model code.	Students implement automated tests. Students run manual acceptance tests. Students maintain 80% statement coverage on all of their code.	Written by students and reviewed by instructors.

CS1: CS1 is an introduction to programming course taught in Java. The learning outcomes for the course cover technical topics related to syntax, flow of control, console I/O, file I/O, arrays, and objects. The course introduces object from the Java API early (e.g. Scanner) but students write their own objects late. The class meets twice a week in a one-hour and fifty minute integrated lecture lab. Course enrollment is capped at 33. In addition to the instructor, there are two (typically) undergraduate TAs in the classroom at all time. Lecture is interspersed with pair and share activities and programming tasks for the students to complete.

There are six programming projects for CS1. The projects are completed individually. Some class time may be allocated to work on the projects, but that varies by instructor and semester. The projects require the students to implement code, following an instructor provided design, to solve a simple problem. Additionally, students are required to Javadoc their source code. The first five projects are solvable in a single Java class and may contain two to five methods in addition to the main method. Most functionality is moved out of the main and the design is intended to implicitly teach students about model-view-controller. Some of the early projects are broken into two parts, where each part requires the implementation of a separate program. The sixth project is an OO project, where students write a program consisting of three to five interacting classes. The project typically has a GUI front end that is provided by the instructors. Students are provided a detailed design and guidance on which classes they should implement first.

Starting with project 3 or 4, students additionally write and submit black box test plans and automated white box tests. Students are provided templates or starting examples/code for both types of test (as appropriate for the project).

CS2: CS2 is a programming concepts course taught in Java. CS2 picks up where CS1 leaves off with a review of OO programming. From there, students learn about advanced OO topic lik inheritance, polymorphism, abstract classes, and interfaces. We then cover the software engineering lifecycle, linear data structures, recursion, finite state machines, and time permitting GUIs, searching and sorting. Class size may range from 60-100 depending on the semester. There are one to two instructors in a given semester with three to four undergraduate TAs. There are three programming projects for CS2. Each programming project consists of three to four iterations. Students are provided the requirements as use cases. In class, the students will inspect the requirements and provide feedback to the instructors. From there the requirements are updated. The first Iteration, Iteration 0, requires that students create a design and black box test plan for the project. With the start of Iteration 1, students are provided a design to facilitate automated grading using an online testing program called Web-CAT[1]. Iteration 2 (and following iterations, if any) build on the earlier iteration by requiring additional functionality or some other modification to the system. For each Iteration greater than zero, students submit their code and unit tests to Web-CAT. Web-CAT runs the instructors tests against the student's code and provides feedback on the syntax of their code and comments using built in static analysis tools. Students also run (and revise) their black box test plan and report the results of the tests. During the last iteration, students evaluate their team and reflect on the project. The reflection includes a comparison of the student's design and the instructor's design for the project.

SE: Software Engineering is a junior level course. The course consists of two 50 minutes lectures and a one-hour and 50 minute closed lab lead by graduate TAs. Each semester of students continues development on a large project called iTrust[2]. iTrust is an online medical records application. The iTrust project has been worked on by 10-12 semesters of students. The class consists of four homeworks and one six-week long project. The first homework is trivial and not associated with iTrust, so it will not be discussed further here. Of the remaining three homeworks, two are paired and the third is solo. Students complete a number of tasks that require delivery of many artifacts from the different phases of development. Each homework is divided into two one-week iterations to help break up the load and mimic the SE lifecycle. The team project consists of requirements elicitation, requirements development, design, implementation, testing, and maintaining the current system. Students present their work to the lab and possibly the class.

[1] Information about Web-CAT is available at: http://www.web-cat.org/.
[2] Information about iTrust is available at: http://agile.csc.ncsu.edu/iTrust/wiki/doku.php.

In Tables 6-9, we describe each of the progressions for a specific genre using complexity analysis across the four courses under analysis. Table 6 describes the complexity analysis of SRS, Table 7 describes the complexity analysis of design expressions. Table 8 describes the complexity analysis of code and comments in the code. Table 9 describes automated testing. These tables will serve as the resources for the analysis covered in Section 5.4.

Table 6. SRS progression

Increasing Complexity Requires	CS1	CS2	SE	Capstone
… longer response	Requirements provided by the instructor, typically in paragraph form.	Requirements provided by the instructor as use cases. Project contains multiple iterations. Number of use cases increases between iterations.	Students enhance an existing system of over 50 use cases. Requirements provided by instructor as use cases for early homeworks. Students write requirements as part of the final project. Typically addition of four to seven use cases. Students consider cross-cutting concerns (logging, privacy, security, etc.).	Students write their own requirements. Student create and revise requirements; often change & developed iteratively.
… more complicated response	Requirements are straight forward and fully defined.	Requirements are straight forward and fully defined. Modeled as use cases.	The situations are complicated with multiple functions and an unfamiliar domain that requires extensive research. Cross-cutting concerns must be addressed.	The situations are usually very complicated, with multiple functions and an unfamiliar domain that requires extensive research.
… response in multiple genres	The requirements are provided to the students in a document that also outlines the design, implementation, and test details.	The requirements separated from the design and implementation details required to complete the project.	The requirements are basis for other genres, like design expression and black box test plan.	The requirements are include with all the major genres in the resulting technical report.
… greater independence of students	The requirements are provided by the instructor.	The requirements are provided by the instructor and inspected by the students before requirements are finalized.	The requirements are provided by the instructor for early homeworks. The requirements elicited from instructor as customer for final project; student requirements incorporated into final SRS.	Students work independently of the instructor, who acts as project manager.
… more students to contribute	Students work individually outside of class.	Students work individually or in two to three member teams outside of class.	Students work in pairs or teams. In closed labs or outside of class.	Teams: 2-4 students. Meetings outside of class time.

Table 7. Design progression

Increasing Complexity Requires	CS1	CS2	SE	Capstone
… longer response	Students provided design.	Design accommodates SRS. Contains low-level diagram and accompanying descriptive narrative.	Design accommodates SRS. Contains at least one diagrams and accompanying descriptive narrative.	Design accommodates SRS and is created iteratively; begins as diagram; evolves to multiple diagrams with accompanying descriptive narrative.
… more complicated response	Students provided design.	Contains low-level diagram and accompanying descriptive narrative.	Design considers cross-cutting concerns and must accommodate existing and new use cases.	Changes throughout development to accommodate need for effectiveness & efficiency.
… response in multiple genres	The design is provided to the students in a document that also outlines the requirements, implementation, and test details.	Design accommodates SRS.	Design accommodates SRS and serves as roadmap for code development.	Design accommodates requirements and serves as roadmap for code development.
… greater independence of students	Students provided design.	Students submit their own design; instructor evaluates design. Students implement the instructor design. Students reflect on the two designs.	Students create design independent of instructor; instructor evaluates design.	Students required to create designs independent of instructor; must defend design decisions when presented to instructor.
… more students to contribute	Students work individually outside of class.	Students work individually or in two to three member teams outside of class.	Students work in pairs or teams in closed labs or outside of class.	Design processed & agreed upon by entire team; provides vision for all & launching point for individual implementation tasks.

Table 8. Code and comments in code progression

Increasing Complexity Requires	CS1	CS2	SE	Capstone
… longer response	Implementation of very small program based on SRS, and design; coding & commenting conventions are defined.	Implementation of small program based on SRS and design; coding & commenting conventions are defined.	Modification and extension of existing medium system based on problem, SRS, and design; coding & comment conventions are defined.	Implementation of entire system based on problem, SRS, and design; coding & commenting conventions are defined.
… more complicated response	Code updated as testing reveals issues.	Code updated as testing reveals issues; second iteration involves extension of existing system.	Modification and extension of existing medium system. Code updated iteratively in final project and as test reveals issues.	Code updated as SRS and design are developed iteratively and as testing reveals issues.
… response in multiple genres	Implemented system must accommodate original SRS.	Implemented system must accommodate original SRS.	Implemented system must accommodate original SRS.	Implemented system must accommodate original SRS.
… greater independence of students	Code & comments done independently, but reviewed by instructors.	Code & comments done independently, but reviewed by instructors.	Code & comments done independently, but reviewed by instructors.	Code & comments done independently, but reviewed by instructors.
… more students to contribute	Students work individually outside of class.	Students work individually or in two to three member teams outside of class. Groups participate in code review.	Students work in pairs or teams in closed labs or outside of class. Groups participate in code review.	Team participates in code review.

Table 9. Testing progression

Increasing Complexity Requires	CS1	CS2	SE	Capstone
… longer response	Black box test plan and automated unit tests.	Black box test plan, unit tests, & code reviews. 100% method coverage of model code.	Comprehensive testing required: unit tests, acceptance tests & code reviews. 80% statement coverage.	Comprehensive testing required: unit tests, acceptance tests & code reviews.
… more complicated response	Black box test plan and automated unit tests.	Black box test plan, unit tests, & code reviews. Unit tests use 3rd party testing framework.	Comprehensive testing means that multiple types of testing and testing frameworks are utilized.	Comprehensive testing means that multiple types of testing and testing frameworks are utilized.
… response in multiple genres	Tests must verify and validate correctness of system implementation with SRS.	Tests must verify and validate correctness of system implementation with SRS.	Tests must verify and validate correctness of system implementation with SRS.	Tests must verify and validate correctness of system implementation with SRS.
… greater independence of students	Templates of black box test plan and automated unit tests provided. Students write tests; evaluated by instructors.	Test frameworks selected by instructor. Tests developed individually or in pair/team; evaluated by instructors.	Test frameworks selected by instructor. Tests developed in pair/team; evaluated by instructors.	Test frameworks used and test code and documents are chosen and developed individually, but reviewed by instructors.
… more students to contribute	Students work individually outside of class.	Students work individually or in two to three member teams outside of class. Groups participate in code review.	Students work in pairs or teams in closed labs or outside of class. Groups participate in code review.	Code is tested by team members other than coder; code reviews are done by team members.

5.4 Step 4: Revisions to curriculum

Using the progressions for common genres from the software development lifecycle, we can identify potential revisions to the curriculum. For example, we may want to add assignments or resources about a given genre to a course to fill a gap in the progression. We may also want to adjust how we use a genre at a particular level. The current progressions listed in Tables 5 through 9 already demonstrate a significant revision of our CS1 and CS2 courses to fill in gaps in the curriculum.

In filling the curriculum gaps, we identified that one potential for difficulty is that different terms were used for a same genre in different courses. The inconsistency of vocabulary may confuse students and lead to incorrect ideas about important genres. One of the major initiatives was to use a common vocabulary across all of the stakeholder classes. A meeting of all interested parties discussed the terms that we could use to describe genres within the software development lifecycle.

When studying the progression of genres across the curriculum, we found that the project write-ups in CS1 were lacking in simplified versions of many of our genres and the terminology used was not consistent with terminology used in the capstone. We have adapted our CS1 offering so that most of the genres are presented to the students in a consistent manner and provide simplified examples of those genres in the field at a level appropriate for the students. While we still maintain a single assignment document for each programming project in the course, the document is now subdivided into sections that are small examples of the problem statement, SRS, design expression, and testing genres. The design is provided for the students, but the instructor follow good design practices, in particular separating logic code from user interface code. When instruction reaches the point of the model-view-controller (MVC) pattern, the instructor can refer to earlier designs that help students accept the pattern. Additionally, students are provided numerous examples, especially of testing resources in both a black box test plan and for automated white box tests.

The CS2 course has a learning outcome regarding the software development lifecycle, but many of the genres were lightly covered in the course and more through example than practice. Identifying these gaps in the CS2 course have allowed for revision of the course structure to allow for a deeper exploration of key genres at a level appropriate for second semester programming students. In our current CS2 offering, students are expected to work with writing their own versions of most of the genres. For each of the projects, students are exposed to SRS in the form of use cases. Students inspect the requirements for clarity and consistency. As part of the requirements inspection report, students write a one to two sentence problem statement. From the requirements, students create their own design expression that includes a UML diagram showing the objects and relationships in a system. The design expression includes a rationale to justify their design. Students create a black box test plan from the requirements before implementing a line of code. During the development phase of the project, students implement the instructor's design. This allows for students to work with a consistent design that will allow for implementation success and also facilitates automated grading of student work. Students write their own automated unit tests using a 3rd party library.

All code in CS2 should be documented and students generate their own API documentation from the Javadoc in their code. Comments are graded on style and on content. Upon completion of the projects, students compile a survey of several reflective questions about the project. One of the questions asks the students to compare and contrast their design with the instructor's design.

However, improvement in CS2 is ongoing. One current gap in the CS2 course is that problem statements are minimally used. While students are required to summarize the project in one to two sentences as part of the requirements inspection, the instructor does not provide a full problem statement for the project as an example. Future work will include incorporation of more example problem statements.

The SE offering did not require much modification. We identified a gap where in some semesters students did not write their own SRS for the team project. Writing an SRS as part of the team project is now incorporated into most offerings of SE. To manage complexity, the instructor provides a final version of the SRS, but input is considered from the best of the student's contributions.

Another gap in SE that will be addressed in future work is the lack of a graded deliverable for maintaining and updating the system design, especially for the six week team project. The design is completed in one of the early iterations, and changes to the design that are made during implementation are not required to be documented. Adding design modification as a graded deliverable to the SE class would provide additional practice on maintaining design expressions.

5.5 Step 5: Assessment

Assessment to evaluate the effect of changes in the curriculum on the performance of students is essential for verifying that our modifications are of value. Most of the effort in adapting the curriculum has occurred in the 2010-2011 and 2011-2012 academic years. Students in the capstone course were surveyed in the Fall of 2010 and that survey helped us identify that students are not seeing or using genres in the CS1 and CS2 courses. We will run the survey in the 2012-2013 academic year's capstone classes to observe if there is a measurable difference in students reporting their usage of various genres in CS1 and CS2.

6. CONCLUSION

We have described and presented an example of a process for enhancing CS/SE students learning by integrating communication in the curriculum. This process is based on the idea that communicating in the genres of a discipline promotes learning of both technical and communication skills in the discipline [2], [3], [4]. The apprenticeship model we have applied here suggests that to learn effectively, students should participate as fully as appropriate in the activities of experts in the field under the guidance of those experts [9]. In the academy, it is the professors in each discipline who function as the experts, providing students instruction and experiences designed to develop the technical and communication abilities of the disciplines. In CS/SE, this means creating learning situations similar to those students will encounter as professionals in the field, situations in which technical and communication deliverables are integrated.

This integration could elevate the profile of communication in CS/SE. It becomes a field in which excellence in communication is valued along with excellence in technical skills. Changing the perception of CS/SE to a discipline that values communication along with technical skills, one that stresses social interaction rather than individual work, could have the effect of attracting students who are more comfortable with that approach, such as women and underrepresented minorities.

ACKNOWLEDGMENTS

This work was funded by NSF CPATH-II Award CCF-0939122. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

REFERENCES

[1] Bates, F. E. and Connor, D. A. Industry Survey for the University of Alabama at Birmingham (UAB): 2005 Electrical Engineering Curriculum Survey. Proceedings of the 24th Annual Frontiers in Education Conference. (2-6 November, 1994), pp. 242-255.

[2] Carter, M. Ways of Knowing, Doing, and Writing in the Disciplines. College Composition and Communication. Vol. 58, No. 3, (February, 2007), pp. 385-418.

[3] Carter, M., Ferzli, M. and Wiebe, E. Teaching Genre to English First-Language Adults: A Study of the Laboratory Report. Research in the Teaching of English. Vol. 38, No. 4, (May, 2004), pp. 395-419.

[4] Carter, M., Ferzli, M. and Wiebe, E. N. Writing to Learn by Learning to Write in the Disciplines. Journal of Business and Technical Communication. Vol. 21, No. 3, (July 2007), pp. 278-302.

[5] Carter, M., Gannod, G., Burge, J., Anderson, P., Vouk, M. and Hoffman, M. Communication Genres: Integrating Communication into the Software Engineering Curriculum. Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training, (22-24 May, 2011), pp. 21-30.

[6] Ford, J. D. and Riley, L. Integrating Communication and Engineering Education: A Look at Curricula, Courses, and Support Systems. Journal of Engineering Education, Vol. 92, No. 4, (2003), pp. 325-328.

[7] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1994.

[8] Henderson, K. Educating Electrical and Electronic Engineers. Engineering Science and Education Journal. Vol. 6, Issue 3, (1997), pp. 95-98.

[9] Lave J. and Wenger, E. Situated Learning: Legitimate Peripheral Participation. Cambridge: Cambridge University Press. 1991.

[10] Miller, C. R. Genre as Social Action. Quarterly Journal of Speech. Vol. 70, Issue 2, (1984), pp. 151-167.

[11] Orr, T. Genre in the Field of Computer Science and Computer Engineering. IEEE Transactions on Professional Communication. Vol. 42, No. 1, (1999), pp. 32-37.

[12] Paretti, M. C. Teaching Communication in Capstone Design: The Role of the Instructor in Situated Learning. Journal of Electrical Engineering. Vol. 97, No. 4, (2008), pp. 491-503.

[13] Pinelli, T. E., Barclay, R. O., Keen, M. L., Kennedy, J. M. and Hecht, L. F. From Student to Entry-Level Professional: Examining the Role of Language and Written Communications in the Reacculturation of Aerospace Engineering Students. Technical Communication. Vol. 42, No. 3, (1995), pp. 492-503.

[14] Reave, L. Technical Communication Instruction in Engineering Schools: A Survey of Top-Ranked U.S. and Canadian Programs. Journal of Business and Technical Communication. Vol. 18, No. 4, (2004), pp. 452-490.

[15] Riley, L. A., Furth, P. M. and Zellmer, J. T. Assessing Our Engineering Alumni: Determinants of Success in the Workplace. Proceedings of the 2000 ASEE/Gulf-Southwest Sectional Annual Conference. (2000), Section 73A1.

[16] Sageev, P. and Romanowski, C. J. A Message from Recent Engineering Graduates in the Marketplace: Results of a Survey on Technical Communication Skills. Journal of Engineering Education. Vol. 90, No. 4, (2001), pp. 685-697.

[17] Vest, D., Long, M. and Anderson T. Electrical Engineers' Perceptions of Communication Training and Their Recommendations for Curricular Change: Results of a National Survey. IEEE Transactions on Professional Communication. Vol. 39, Issue 1, (March, 1996), pp. 38-42.

[18] Wolfe, J. How Technical Communication Textbooks Fail Engineering Students. Technical Communication Quarterly. Vol. 18, No. 4, (September, 2009), pp. 351-375.

APPENDIX

Below is the survey given to students in the capstone course. For each activity listed in the survey, students would circle the course number if they did the activity in the course. Students would then indicate the importance of the activity in the course on a scale of low, medium or high. Students would also rate their confidence level in performing the activity. The Answer Sheet for the first question is provided.

ANSWER SHEET

If you have done the activity, **circle the course or courses** you did it in. If other, **fill in course number**. To indicate the level of importance of the activity in the course, **circle low, medium, or high**.						**Circle** your confidence level in performing this activity
1.	CS1 L M H	CS2 L M H	Data Struct. L M H	SE L M H	Other: _____ L M H	L M H

CSC 492 – Fall 2010 - Survey

To help us provide instruction you need to succeed in this course, we would like to know about your experience with the kinds of activities you may be doing this semester. On this sheet, you will find a list of various activities associated with computer science and software engineering. For each of these activities, fill in your answer on the answer sheet.

1. Have you written a problem statement (defining a problem to be solved by software in order to meet the needs of an end-user)?

2. Have you presented a problem statement orally?

3. Have you written requirements (description of what a program should do)?

4. Have you analyzed requirements to determine what you know and what you need to know in order to implement the requirements in a program?

5. Have you inspected and corrected requirements (your own or others')?

6. Have you reviewed requirements to verify whether or not they meet users' needs?

7. Have you revised requirements so that they better meet users' needs?

8. Have you written a design that translates requirements for implementation?

9 Have you reviewed a design to determine if it will accomplish the requirements?

10. Have you revised a design so that it will better accomplish the requirements?

11. Have you written a program based on a design?

12. Have you reviewed a program to find errors of syntax and semantics?

13. Have you corrected errors of syntax and semantics in a program?

14. Have you read a program to determine what it is supposed to do?

15. Have you revised a program so that it is better able to do what it is supposed to do?

16. Have you modified a program for a different use?

17. Have you inspected a program to ensure that it conforms to requirements and design?

18. Have you continued writing a program that had been begun by someone else?

19. Have you written a narrative description of code, including a list of file names or directories included?

20. Have you written a developer guide?

21. Have you written a user guide?

22. Have you written a plan for testing a program?

23. Have you presented orally a plan for testing a program?

24. Have you written a report of the results of a test of a program?

25. Have you presented orally the results of a test of a program?

26. Have you revised a program based on the results of a test of that program?

27. Have you read a technical article in the field of computer science?

28. Have you summarized or analyzed a technical article in the field?

29. Have you used technical articles in the field to enhance your knowledge to help answer a question or solve a problem?

30. After using technical articles to answer a question or solve a problem, have you reported out (either orally or in writing) on that activity?

31. Have you worked on a team that developed ground rules used to guide your team's work approach?

32. Have you worked on a team that defined roles so that expectations of team members were clear & followed?

33. Have you worked on a team that created meeting agendas and minutes?

34. Have you worked on a team that regularly facilitated meetings so that everyone contributed to the meeting?

35. Have you been on a team that worked on consensus development?

36. Have you worked together on a team to develop an audience-sensitive oral report?

37. Have you worked together on a team to develop an audience-sensitive written report?

38. Have you worked on a team that created specific action items for each member who was then held accountable for those items?

39. Have you worked on a team that freely gave feedback to one another?

40. Have you worked on a team where all team members had a general understanding of how your project or assignment worked and how their individual piece specifically contributed to the project or assignment as a whole?