SUMMARY
Active-learning exercises are an effective use of class time to bring about desired learning outcomes. Instead of listening to a lecture, students are engaged in tasks that allow them to discover new knowledge, or apply what they have just learned. A barrier to wide usage of active-learning exercises is the need to design them, since few are available in textbooks, technical papers, or on the Web. The work reported in this paper demonstrates that students can design active-learning exercises that are worthy of being used in CS1 and CS2. This frees the instructor from having to write all the exercises him/herself. This paper makes three contributions: a methodology for creating student-generated active-learning exercises, several exercises for teaching difficult concepts in CS1 and CS2, and guidance about the kinds of active-learning exercises that students will enjoy and learn most from.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and information systems education, Curriculum.

General Terms
Algorithms, Languages.

Keywords
Active learning, Kinesthetic learning exercises, Peer review

1. INTRODUCTION
Computer-science instructors in growing numbers are coming to realize that a straight lecture format is not the best way to teach programming. Students’ attention spans are far shorter than the usual 50- or 75-minute lecture. Competition from laptops and text messaging is increasing. Some instructors have responded by banning laptops, but it seems ironic to ban computers from a programming class.

Active learning is a good alternative. The instructor

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE ’09, March 3–7, 2009, Chattanooga, Tennessee, USA.
Copyright 2009 ACM 978-1-60558-183-5/09/03...$5.00.

lectures in short bursts, say, 10 minutes at a time. In between, students work with each other on activities that help them discover new principles, or solidify principles just learned. Meta-analyses [9, 10] show that cooperation is much more effective than individual learning. Many presentations at SIGCSE [1, 2, 3, 5, 7, 11, 16, 17] have focused on active learning. Inroads published a four-part series on active learning [12–15] by Jeff McConnell. These articles can serve as a road map for integrating active learning into the CS curriculum. Yet any instructor who tries to build each class period around active-learning exercises will face a formidable hurdle: Designing good active-learning exercises is not easy, and there are many topics in CS1 and CS2 for which no published active-learning exercises exist.

Our belief is that students, after experiencing a dozen or more active-learning exercises, are capable of designing their own exercises on other topics. If this hypothesis is true, not only would it free the instructor from having to create all the exercises; the students, by creating exercises, would deepen their knowledge of the topic. It is even plausible that students’ minds are better attuned to their peers’ interests, and therefore capable of designing more engaging and relevant exercises than instructors could.

2. THE METHODOLOGY
This experiment was carried out in NCSU’s CSC 216: Programming Concepts—Java. This is the second course in a three-course Java programming sequence, and contains some topics (e.g., debugging and testing) that are traditionally in CS1, and others (e.g., linked lists, binary search trees) that are traditionally in CS2. Twice during the Spring 2008 semester, students were assigned to create an active-learning exercise over the material covered in class. They were given 30 to 40 minutes of class time to get started on it, and then a week to complete it. These exercises were submitted on a wiki page, and then reviewed by the instructor and peer-reviewed by other students, using the Expertiza system [6]. Students were then given a few days to revise their exercises in response to these comments. A second round of review ensued, during which the peer reviewers were asked to check that the authors had addressed the points raised by the instructor and themselves in the earlier round of review.
Five of the student-authored exercises were later used in CSC 216. Four of the exercises created during the spring were used in the Summer 2008 section. One of the exercises, because it covered a topic that had not yet been covered in class, was able to be used in Spring 2008.

3. SAMPLE EXERCISES

Many of the students put a lot of effort into their exercises, creating diagrams, props, and even videos (for which they were given extra credit). The videos will be helpful not only for students, but also for instructors to get a quick idea of how the exercise will play out in their class. To give an idea of the variety of exercises that were created, here are synopses of some of the best.1

3.1 Debugging

Beginning programmers don’t appreciate the effort that is required to debug a program. This exercise effectively illustrates the difference between giving “English-like” instructions and actually coding them in a program.

A program is represented on a sequence of cards. On the front of each card is an English description (a “comment”) of a step in the program. On the back is a statement, or a very short sequence of statements, that is supposed to implement the step. This is a kinesthetic learning activity [19] that involves one student acting as the programmer and a set of students who hold one card each. The students are arranged in a line, representing the sequence of steps in the program. They pass a ball (or other prop) between them to represent movement of the program counter.

The students stand with the comment side of the card facing the programmer. The programmer passes one or two parameters to the first step of the program. The student with the first card “executes” the statements, and tells the next student in line the value of the variables that his/her code has computed. This process repeats until all cards have been “executed,” and the programmer is then told the result. If the result is what the programmer expected, all is well. But if not, the students go into debug mode and re-execute the program, with the programmer being able to query the values of the variables after any step (just like a debugger does). Eventually, the bug will be located.

3.2 Copying objects

Students often have difficulty grasping the difference between the various ways of copying objects. This exercise uses props to illustrate the difference.

1. Obtain several pieces of paper (any type), a block of sticky notes, a bundle of string, a pen or pencil, and some clear tape. These objects will represent classes, member variables, and references, respectively. The tape is simply to hold the yarn in place.

2. Cut several pieces of yarn approximately three feet long, and set them aside.

3. Label a piece of paper with the desired class name, and create several variables for the class by labeling sticky notes with desired variable names and sticking them to the paper.

4. Have the students illustrate a shallow copy by creating another piece of paper (with the same name) and, rather than using sticky notes for the variables, simply connecting the new object to the variables of the old object using the string and tape (do not tape the ends of the string to the old object variables, however).

5. At this point, is should be clear to the entire group that the variables do not actually belong to the new object, but are simply referenced by it. To illustrate this point even more fully, remove the old object, leaving the new object with reference strings which point to nothing. This illustrates what happens when the original object is reclaimed in memory.

6. Prompt the class for ideas on how to alleviate this problem, and if desired, illustrate this deep copy by bringing back the original object and creating new sticky note variables for the new object.

3.3 Sorting

Selection sort. Many activities devised by the students involved various sorting algorithms. To illustrate selection sort, one group of students suggested handing out a single playing card to each of a group of 7 (or 15) students. Or, students could be given blank cards and asked to write, e.g., their name or birthday on the cards. The students then line up in unsorted order. The instructor hands a tennis ball to the first student. Each student down the line compares the value of his/her card with the value of the ball-holder's card. If a student has a card lower in value than the ball-holder's, the ball-holder should toss the ball to that student. Once the end of the list is reached, the student holding the ball trades places with the first person in the line. That student should stand slightly away from the rest of the "list," and not participate in the remainder of the sorting process. The student who has just been removed from the unsorted line should pass the ball to the next person in line.

This process is repeated until the entire list is sorted.

Quicksort. A similar exercise illustrates quicksort. Students again start with cards containing an alphabetic or numeric value. The first student in the randomly-ordered line serves as the pivot. Students with values less than the pivot form a list. The rest of the students (minus the pivot) form the greater-than-list. These two new lists perform quicksorts on themselves, and this process continues recursively until the lists contain only one element. Then they merge together, with the less-than list coming first, followed by the pivot, followed by the greater-than list.

---

1 All of the exercises described in this paper, and more, can be found at http://wikis.lib.ncsu.edu/index.php/SGALE.
Mergesort. Hierarchical mergesort can be illustrated by seven students, each wearing a paper hat, either a “king” hat, a “delegate” hat, or a “peon” hat. There is one king, two delegates, and four peons. The students cooperate to sort a list of eight elements. The king splits the list in half, and gives each four-element lists to a delegate. Each delegate splits the list in half again, and gives each two-element list to a different peon. The peons sort their two-element lists, and hand them back to their delegate, who merges them into a four-element list. The king then takes the four-element lists and merges them into a (sorted) eight-element list.

The mergesort and quicksort groups each produced videos, and suggested that their video be shown to the class before the exercise was carried out.

3.4 Games
Several of the exercises were takeoffs on board games or television game shows. Two examples are given below.

One group created Javadash, a takeoff on Balderdash. The rules closely match the real game, except that the class is broken up into teams, with each team writing a definition for the term in question. The “official” definition is also included in the set. Then each definition of the term is read aloud, and students choose which is the official definition. Terms may include items like “super,” “abstract method,” “protected,” and “clone.” Each team member gets a point for each time the team’s definition was chosen. In addition, each player who chose the correct definition gets a point.

“Are You Smarter than a Computer Science Major?” mimics “Are You Smarter than a Fifth Grader?” One student in the class is chosen as the “contestant” and is asked to come forward. The rest of the class act as the “computer science majors.”

The contestant will then choose a classmate to be his/her helper. After this is completed, the contestant will receive a series of questions which (s)he must answer in order to progress in the game. A total of 10 questions will be posed, two each on five selected topics. The contestant selects one of the five topics. Once each question is asked, all of the “computer science majors” including the helper answer the question. Once the contestant chooses an answer, it will be shown on the PowerPoint slide. At this point, the rest of the students will show their answers so everyone will be able to compare the contestant’s answer with the computer science majors’ answers, and see if the contestant is smarter than a computer science major.

In addition, the contestant will have a “cheat” to help with difficult questions. This is the part of the exercise that demonstrates exception-handling. If the contestant does not know the answer to a question, (s)he can “throw” a tennis ball to the helper, who acts as an “exception handler.” The handler then “catches” the ball and has the opportunity to answer the question for the contestant. If the handler answers the question correctly, the contestant continues in the game. This can be done three times.

If the contestant is correct, (s)he progresses to the next question, until all questions have been answered. The contestant’s turn ends when either all questions have been answered, or a contestant misses a question. The students who created this exercise also made up two sets of ten questions each, and a video demonstrating how the game is played.

3.5 Videos
Some students created what are essentially training videos along with self-study exercises. These would probably not be used during class, unless the class meets in a computer lab, but they still qualify as active learning, because students learn by doing. One group produced a video on debugging in Eclipse, and another group did one on how to use Eclipse to submit to the Web-CAT automated grading system [20].

4. STUDENT REACTION
In addition to the regular university course evaluation, four surveys were administered at the end of the spring semester. One survey asked about students’ reaction to active-learning exercises; a second asked students to rate each of the active-learning exercises used by the instructor; a third asked using a wiki for collaborative work; and the last asked about the peer-review process and software. Probably because of the multiple surveys, response rates tended to fall off on the latter surveys (they were 52%, 78%, 30%, and 30% respectively, with n = 50). In the summer semester, only the survey rating the active-learning exercises used in class was administered; response was 88% (n = 16).

One of the most striking findings was that active learning tended to increase attentiveness during class. On a Likert scale of 1 to 5, with 1 being “strongly agree” and 5 being “strongly disagree,” students gave a mean rating of 2.6 to the statement, “In classes that did not involve active-learning exercises, I tended to be distracted (e.g., daydream, do unrelated work, or read my e-mail) during class.” To the same question, asked about classes that did involve active learning, they gave a mean score of 3.4. The difference was significant at the 99% level (p = 0.0047). The students were mildly positive on the question, “I learned a lot from doing the active-learning exercises” (2.6) and more positive on, “I enjoyed doing the active-learning exercises” (2.3). The students were allowed to submit prose comments on any question; most of the negative comments were from students who claimed they already knew the material before the AL exercise was presented.

Students were only neutral on whether they learned a lot from devising active-learning exercises (2.9), but mildly
positive on whether they enjoyed doing so (2.6). The students who were positive mentioned the opportunity to be creative and work in a group; those who were negative said that programming was more challenging, or that they did not want to teach, and thus didn’t derive a benefit.

Students were very positive on the wiki; with a mean score of 1.4, they agreed that “using a wiki made it easy to collaborate with other students. They were also very positive on the feedback from their reviewers, scoring 1.9 on the question, “The feedback I received helped me to improve my work.” However, they did agree that “there were too many deadlines” for reviews and resubmission (2.3). Several of the prose comments said it would have been helpful to assign the active-learning exercises earlier to give students more time to revise their work (both exercises were done in the final five weeks of the semester).

5. COMPARING EXERCISE TYPES

5.1 Student- vs. instructor-generated exercises
At the end of the semester, students were asked to rate each of the active-learning exercises on a scale of 1 to 5, with 5 being high. They were asked how much they learned by doing each exercise, and how much they enjoyed the exercise. In Spring 2008, the students rated 13 instructor-generated exercises and 1 student-generated exercise. In Summer 2008, students rated 11 instructor-generated exercises and 4 student-generated exercises.

In Spring 2008, the average score for learning on all 14 exercises was 3.1, and the average score for enjoyment was 2.8. The student-generated exercise (mergesort, described in Section 3.3) rated 3.2 for learning and 3.0 for enjoyment, which is slightly above average by each measure; however, the difference between instructor-generated and student-generated exercises was not significant (p=0.69 for learning, and p=0.34 for enjoyment)

In Summer 2008, the average score for both learning and enjoyment was 3.5. The student-generated exercises averaged only 2.9 for learning, but 3.8 for enjoyment. In fact, two of the four student-generated exercises were rated the most enjoyable of the semester, while two of them were tied for the lowest score on learning! This apparent paradox is easily explained by the fact that three of the four student-generated exercises were games (Javadash, Family Feud, and Are You as Smart as a Computer Science Major?) designed as review exercises (the fourth was a soda-pop machine as an example of a finite-state machine). Thus, we should not read too much into the fact that the student-generated exercises were rated lower in learning (significant at the 99% level, p=0.00006) but more enjoyable (significant at the 90% level, p=0.057).

5.2 Exercise formats
Most of the active-learning exercises in Summer 2008 involved writing code. Students were given an opportunity to submit the code for credit. In Spring 2008, by contrast, there was no opportunity to receive credit for doing an active-learning exercise; this may have been a factor in the higher perceived learning by the summer students. One exercise in both spring and summer involved o-o class design; it was rated near average in learning (3.0, 3.2) but well below average in enjoyment (2.5, 2.5). This is unfortunate, because o-o design is such an important topic; it may be that students need more programming experience before they have a clear idea of the responsibilities of various classes in a program.

The other exercise format that was most disliked were the two spring exercises that involved browsing the Java class library for examples. One exercise asked students to find an interesting interface and describe it to the class; the other asked them to find an error or exception class. These were rated 2.9 and 2.6 for learning, but only 2.3 for enjoyment.

At the other end of the scale were exercises that asked students to experiment with a simulation to discover properties. One of these had students fill in the missing statement in a JHavéPop exercise on linked-list programming. The other used the Hope College Sort Animator [8] to deduce the complexity of different sorting methods, and the effect of initial key distribution. These exercises rated 3.6 and 3.9 for learning, and 3.4 for enjoyment.

This suggests that future active-learning exercises might well be designed around algorithm visualizations, like the ones available on the Algoviz wiki [18]. If class is held in a regular classroom, that should not be a barrier; just ask students to bring their laptops to class. Of course, not everyone can or will, but students can be asked to pair up (thereby introducing another form of collaborative learning).

Kinesthetic learning activities are often thought to be popular, but the five KLAs in our sample were only near average (3.1 for learning, 3.0 for enjoyment in the spring semester). So while useful, KLAs should not automatically be favored for future exercises.

6. DISCUSSION
A large body of evidence suggests that active learning is more effective than lecturing. Our study highlights one particular aspect of this effectiveness: Students reported feeling significantly more attentive in classes where active learning was used.

Our preliminary evidence suggests that student-generated active-learning exercises are effective alongside instructor-generated exercises. There was no significant difference in learning or enjoyment between instructor-generated and student-generated exercises in the spring semester. In the summer semester, there were significant differences, which
can probably be attributed to the fact that most student-designed exercises were games.

Furthermore, the study uncovered several avenues for improving student-generated exercises: (i) encourage exercises based on pre-existing interactive algorithm visualizations; (ii) have students develop questions on additional topics for games made up by previous students; and (iii) obtain feedback on the strengths and weaknesses of particular exercises and have subsequent groups of students work to address the shortcomings. Working together, student-generated exercises hold promise for improving learning, by helping each class to “stand on the shoulders” of earlier cohorts of students.

7. ACKNOWLEDGMENTS

Jennifer Carter and Jonathan LeRay designed the debugging activity. The shallow/deep-copy exercise was created by Mark Etheridge and Nick Forrer. The selection-sort exercise was devised by Jonathon Harding, and Alex Klaus; quicksort and mergesort were done by Tyler Arehart, Ryan Hasian, and Alexander Innes.

David Bleaking and another student (who requested anonymity) created Javadash. David Wright and Robert Hudson came up with “Are You Smarter than a Computer Science Major.” Rob Udechukwu and Gabi Ghali created the Eclipse debugging video. The authors wish to thank Kristy Boyer for her very helpful comments on the draft.

The Expertiza project is funded by the National Science Foundation under award 0536558. The work on using peer evaluation and videos to review active-learning exercises is funded by the NCSU/Duke Center for Advanced Computing and Communication, through the sponsorship of the William and Ida Friday Institute for Educational Innovation.

REFERENCES


