Techniques for Active Learning in CS Courses

Tom Briggs
Department of Computer Science
Shippensburg University
thb@ship.edu

Abstract

Studies show that active learning promotes improved long-term retention of course material in students. Research suggests that CS students tend to have learning styles that make an active environment almost critical to their successful mastery of material. This paper demonstrates the effects of a novel lab experience in an objects-first Computer Science 2 course. Our lab is a novel departure from traditional lab based courses, in that it promotes student self-learning and interleaved into the class lectures. We show through assessment comparison that an active learning improves student grades, comprehension, and satisfaction with the course.

INTRODUCTION

Active learning is a common technique used to improve students comprehension and retention of material [1], [4]. The most common application of active learning described in the CS education literature is in introductory programming courses [3], [7]. In these introductory courses, techniques such code review, debugging sessions, problem solving, and peer collaboration are commonly used to show improvement in the comprehension and long-term retention rates of material. Research shows that students involved in an active learning environment demonstrate a higher rate of retention and mastery of the material and lower drop out rates [3]. Another study shows that active learning promotes as much as a 70% increase in the amount of long-term retention of knowledge [4]. Students of most disciplines demonstrate better mastery of the material when they are able to interact with it. Research in learning styles assessment suggests that it is especially true for computer science students [6]. We will show how a novel lab experience and other classroom modifications can create an active learning environment and lead to an improvement in student outcomes.

Felder-Silverman Index of Learning Styles

The Felder-Silverman Index of Learning Styles breaks learning styles into four groups [5]:

- Active and Reflective
- Sensing and Intuitive
- Visual and Verbal
- Sequential and Global

Active learners prefer an environment that enables them to learn by using the knowledge such as writing programs or discussing material with their peers. Reflective learners prefer an environment that enables them to cogitate over the material. Sensing learners prefer learning facts and concepts, intuitive learners prefer learning possibilities, applications, and relationships. Visual learners prefer learning from material they can see: charts, figures, and demonstrations. Verbal learners prefer words, either spoken or written. Sequential learners follow material in a step-by-step sequence. Global learners tend to learn by putting material into a global context and seeing how the material relates, and then they will “get it.” [5].

In [6], Thomas, Ratcliffe, et.al, report the results of administration of the Felder-Silverman survey to 107 students in an introductory CS course. The results show a near even split for sensing/intutitive and sequential/global. There was a tendency towards active learning (55%) over reflective (45%). The strongest difference was visual (83%) vs. verbal (17%). The results of the survey add a quantitative dimension to the argument in support of active learning. If CS students tend to be active and visual learners, then one of the most effective learning environments is one where they are able to interact with and visualize the material in some fashion. In other words the need an active learning environment.

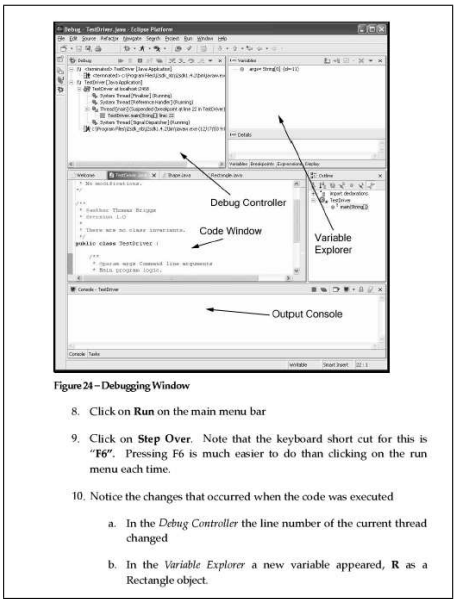


Figure 24 – Debugging Window

8. Click on **Run** on the main menu bar
9. Click on **Step Over**. Note that the keyboard short cut for this is "**F6**". Pressing F6 is much easier to do than clicking on the run menu each time.
10. Notice the changes that occurred when the code was executed
    a. In the *Debug Controller* the line number of the current thread changed
    b. In the *Variable Explorer* a new variable appeared, **R** as a Rectangle object.

Figure 1: Example Lab Exercise Demonstrating Eclipse Techniques

Active Learning Techniques

Active learning does not require a large amount of preparatory work. Felder proposes the following simple approach to incorporating active learning in a class[2]. First, ask the students to interact with the material, for example, ask them to solve part of a problem, derive a piece of code, sketch out a proof. Next, ask them to break up into groups, and tell them how long they have to solve the problem. Finally, after the time is expired, call on a few groups to share their solutions.

The important part of this strategy is that incorporating this break in the flow of the class allows students to refocus their attention. Students are able to interact with their peers and try out their conceptual view of the material and get immediate feedback on their understanding of the material. Finally, pairing with peers helps academically weak students begin to comprehend. Felder suggests that even a five minute active period during a 50 minute lecture can change the dynamic of the entire lecture and keep students better engaged in the material.

AN ACTIVE APPROACH TO COMPUTER SCIENCE 2

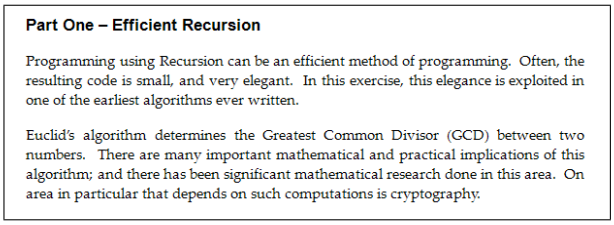
At Shippensburg University, Computer Science 2 (CS2) is four credits and is the second course in the core of our program. The typical student is a freshman or sophomore CS major. The course objectives include using advanced data structures, abstract data types, and object oriented design. The course is taught using Java and the Eclipse IDE in an objects first approach. In objects first, object design, inheritance, abstraction, and design patterns are integrated into to the traditional data structures and ADT material.

The active techniques used in this course include many of the same techniques that appear in the literature such as peer review, paired programming, and small group problem solving [3]. One of the main techniques was a student lab manual that we created to promote an active environment, to fill gaps in the material in the text book, and to deliver material that appealed to a wider range of learning styles. The novel aspects of the lab manual include task based reference sections, interactive lecture/workshop labs, and selected source code examples.

Lab Manual

The lab manual organized additional course material into a single print volume containing source code examples, screen shots, sample output, and guided activities that appealed to the most common learning styles (visual and active) [6]. The lab manual was divided into four primary sections *Technology Manual*, *Reference Section*, *Guided Labs* and *Sample Code*. The final printed manual was 270 pages. The manual was sold by the campus book store as a required text to supplement the text book. The cost of the manual was nominal and recovered the printing and binding fees.

The Technology Manual contained instructions for using the various development environments, computer systems, and off-campus resources. For example, students used Sun workstations in the course, and

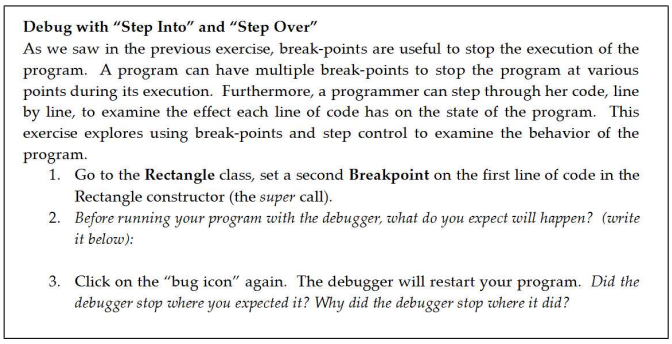


Part One – Efficient Recursion

Programming using Recursion can be an efficient method of programming. Often, the resulting code is small, and very elegant. In this exercise, this elegance is exploited in one of the earliest algorithms ever written.

Euclid's algorithm determines the Greatest Common Divisor (GCD) between two numbers. There are many important mathematical and practical implications of this algorithm; and there has been significant mathematical research done in this area. On area in particular that depends on such computations is cryptography.

Figure 2: Example Introduction from 'Lab 4 - Recursion'



Debug with "Step Into" and "Step Over"

As we saw in the previous exercise, break-points are useful to stop the execution of the program. A program can have multiple break-points to stop the program at various points during its execution. Furthermore, a programmer can step through her code, line by line, to examine the effect each line of code has on the state of the program. This exercise explores using break-points and step control to examine the behavior of the program.

1. Go to the **Rectangle** class, set a second **Breakpoint** on the first line of code in the Rectangle constructor (the *super* call).
2. *Before running your program with the debugger, what do you expect will happen? (write it below):*
3. Click on the "bug icon" again. The debugger will restart your program. *Did the debugger stop where you expected it? Why did the debugger stop where it did?*

Figure 3: Example Instruction from 'Lab 2 - Using Eclipse'

the technology section included instructions using Sun's windowing manager (CDE). The technology section also included detailed step by step instructions for using Eclipse. The instructions were organized by the tasks the students would encounter. Examples include: *Import External "jar" File*, *Recover Previous Versions of Files*, *Automating Getter and Setter Method Creation*.

Labs

The lab manual contained eleven lab exercises and three projects. Each lab corresponded to a chapter of the text book, and augmented the material in the text. The labs followed a progression such that the early labs were very specific and included detailed step-by-step instructions while later labs were very general, and involved more independent problem solving skills. The labs were designed to instruct and challenge student perceptions of the material and to develop students through problem solving and programming skills.

During the lab portion of the class, students worked in teams of two or three to complete the given tasks. Students were encouraged to carry on discussions between themselves and other lab groups, and frequently discussed challenging parts of the material with each other. Individually, students also often used the time to ask questions they were afraid to ask in front of the whole class.

Our labs are a novel departure from the traditional in-class lab. The goal was to interleave description with interaction. Each lab starts with a brief, global overview of the material and important topics to be addressed in the lab (see figure 2). Each step of the lab presents the student with some background narrative or sample code, and asks them to actively engage with the material in some way, such as solving a problem, implementing some code or predicting the result of some action (see figure 3). Finally, there are a series of exercises at the end of each lab that asks students to further explore and extend the lab material (see figure 4).

The lab manual contained the following labs. Many of the labs included printed versions of complete or partial source code which students had to enter into the computer. This helped keep students organized and on-task, and it also presented the students with examples of well-written source code to use as a reference. The requirement that students manually entered the code (as opposed to downloading source code files) forced them to read the code carefully.

Hello World Introductory lab to get students familiar with environment (Sun CDE).

Using Eclipse Walk-through several features of Eclipse, such as entering code, setting the run-time environment, building and running programs, and using the integrated debugger.

Programming with Files Basic file I/O, including directory access, text file I/O, binary data streams, and object data streams. In part one, students were guided through an exercise to make their own version

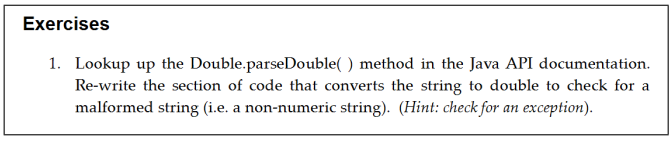


Figure 4: Example Exercises from 'Lab 3 - Input and Output'

of the `DIR` command using the object oriented design, polymorphism, loop iteration, arrays, and string and numeric formatting. In part two, students were guided through an exercise to read and write text files containing descriptions of shapes, and then use that information in a computation. Finally, in part three, students use built-in Java packages to read serialized object data from a file demonstrating the utility of the Object hierarchy.

Recursion Several aspects of recursion, including code simplicity, efficiency, tail recursion, and recursive traversal of a hierarchical data structures. Part one explores the simple elegance of recursion with Euclid's GCD algorithm. In part two, students experience the inefficiency of repeated recursive calls with the Fibonacci sequence. Part three highlights the relationship to certain types of loops and tail recursion by rewriting loops into recursive methods and vice versa. Finally, part four uses the directory listing program from the previous lab and extends it to recursively descend into sub-directories. This last part is interesting because students are familiar with the concept of a directory tree and are able to grasp the effectiveness of recursion to follow this hierarchy.

Object Oriented Design This lab builds on the previously developed simple UML and class diagrams, and asks students to develop UML for a more complex system. Students were introduced to an optional UML diagramming package, Poseidon. The lab presents students with a detailed description of a problem and a related set of business rules, and they work in larger teams to develop the design structure for the problem. At the conclusion of the lab, each group presented its solution, and the other groups (having worked on the same problem) evaluated the solutions.

Containers The purpose of this lab was to make students familiar with the operations defined by the `java.util.Container` class. Students completed code to implement a simple array list data structure and instantiated the abstract methods of the Container class.

Performance The first part of this lab asked students to add benchmarking instrumentation to existing code to clock its run-time. The second part asked students to compare the runtime of their array list that was developed in the previous lab and several other Container objects. The third part asked students to compare the runtime of several different sorting algorithms. In each case, students were asked to collect run-time statistics from each different data structure or algorithm and to create a graph showing the results. These graphs were then compared against plots of the standard growth functions.

Table Driven Programs This lab explored dynamic programming and table driven programs through Floyd-Warshall's All-Pairs Shortest Path algorithm.

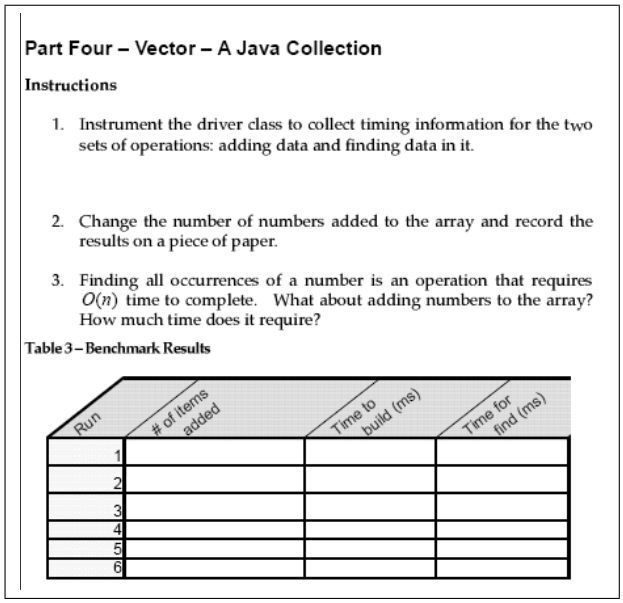
Dynamic Data & Linked Lists Students were guided through implementation of a singly linked list data structure. Students then compared the performance of their implementation to the built-in Java `java.util.LinkedList` and `java.util.Vector` classes. This lab also introduced basic unit testing (using JUnit) as a development aid. This helped students ensure their code was working, and simultaneously, allowed students to use the collection as well as implement it.

Stacks and Priority Queues Students explored using stacks and priority queues by building a basic event simulator.

Maps Students used the `java.util.HashMap` collection class to implement a dictionary lookup / spell checker class.

Active Example - Algorithm Runtime and Performance

The active approach was used to tie the theory of Big-Oh notation to its empirical runtime. One fifty-minute class was used to introduce the material. Topics included introduction to Big-Oh, growth classes, and runtime of common operations. Students were asked to work in small groups to examine pairs of sample code fragments and predict which would be faster. After a few minutes, each group presented its



Part Four – Vector – A Java Collection

Instructions

1. Instrument the driver class to collect timing information for the two sets of operations: adding data and finding data in it.

2. Change the number of numbers added to the array and record the results on a piece of paper.

3. Finding all occurrences of a number is an operation that requires $O(n)$ time to complete. What about adding numbers to the array? How much time does it require?

Table 3 – Benchmark Results

Run	# of items added	Time to build (ms)	Time for find (ms)
1			
2			
3			
4			
5			
6			

Figure 5: Students make empirical observations throughout the lab.

solutions and a brief discussion period followed. The professor facilitated the discussion by encouraging students to explain their predictions, while keeping order during the discussion. The class ended with an instructor lead demonstration of techniques to instrument a program to collect its run-time. The next two class meetings were devoted to the in-class portion of the lab that corresponded to the material.

The first part of the lab instructed students to devise methods to observe and compare the runtime behavior of certain data structures to their theoretical behavior. The second part asked students to code and execute their tests and record their empirical results using their own coded data structures and selected members of the Java Collections API (see figure 5). As an exercise, students were then asked to review their recorded measurements, make graphical plots of their data and then compare these plots with the plots of common Big-Oh functions. In several cases, students found unexpected differences between the theory and the evidence. For example, performance of the 'LinkedList' structure showed larger jumps in run-time than anticipated. Students were directed to the source code appendix of the lab manual to examine relevant source code samples of the Sun Java API to help explain contradictions in their results. They were able to see how professional programmers implemented the same data structures discussed in the course. This type of interaction helped students remain active while challenging them to explain empirical results that were not supported by the theory and help build strong mental models of the material.

At the end of the second lab day, the homework portion of the lab was assigned to be due the following week. The fourth and final day spent on the material was designed to summarize the lab. The professor facilitated a discussion of the observations of the labs and helped students resolve lingering questions. The lecture concluded with a discussion of the fundamental importance of studying the runtime behavior of algorithms and the practical limitations of computability.

Pitfalls of our Approach

Adopting an active learning environment created a dynamic, free and open learning environment. Students quickly adapted to an environment that encouraged them to openly express themselves, rewarded participation, and stimulated their interest in the material. This approach did create other, unforseen, challenges. Here, we discuss these challenges and suggest techniques to avoid them.

The first challenge is to keep everyone involved. Student teams tend to have one "driver" and one "watcher." Experientially, this is especially true when there is a significant difference in the comprehension level of the two students. One technique, borrowed from extreme programming (XP), is to force drivers to switch every ten minutes. Another technique is to make students pair up differently for each lab.

Another challenge is to keep students focused on task. Students quickly adapted to the active environment, and thrived in the open lab experience, where they were able to cooperatively work on a solution to a task or problem. However, there was a tendency for students to drift into unrelated topics. The instructor

Grade	Fall 2003	Spring 2004
A	23%	26%
B	9%	19%
C	14%	26%
D	23%	7%
F	20%	11%
W	11%	11%

Table 1: CS2 Final Grades Comparison

needs to follow the conversation of the student groups, and, when appropriate, gently reel them back into the task at hand.

Absenteeism became a major problem in our environment. Students worked on Sun workstations which required them to log in, and store their files in a students home directory. Inevitably, one of the students who "owned" the files would not be in class, and the other team member would not have a copy of the files. To help mitigate this challenge, attendance penalties were added to later syllabi, and students were required to share files with each of their team mates at the end of class.

RESULTS

The results compare two Computer Science 2 courses, one in Fall 2003 and the other in Spring 2004. Both courses had the same instructor, textbook, classroom resources, similar enrollments (34 and 42 respectively) and student backgrounds. One difference is that four students in the Spring were repeating a failure from the previous Fall. The conventional teaching methods used in the Fall such as lectures, labs, and homework assignments that worked well for traditional structured programming were not well suited for the objects first approach. Students in the course were frequently frustrated with the complexity and scope of the material, and the general tone of the course was largely negative.

An end of term post-mortem analysis of the results of the Fall 2003 course showed that there were problems. The student's final grades (Table 1) show that nearly one-half of the students either dropped out (W), failed (F), or received a poor grade (D). Student feedback, collected through a standard University-wide 'student satisfaction' survey, included statements such as "projects were too large and complex" or "spend less time covering logic and more time programming."

A similar end of term post-mortem of the Spring 2004 course which relied on the active techniques shows a moderate improvement in the final grades. Student feedback is the clearest endorsement. One student responded that the most positive aspect of the course is that the "labs get really involved and really test your skills" and another wrote "this is more hands on and challenging than any other CS course I've taken previously, which really helps [me] further understand the subject matter."

The active learning environment in Computer Science 2 helped transform the class into an interesting, challenging, and interactive environment. The effort that was put into reworking the course and assembling the manual was well worth the end results. This technique was effective, and well liked by the students.

CONCLUSIONS

Active learning environments are an effective style of teaching. Research suggests that active learning is especially effective for CS students who tend to be visual/intuitive learners. Techniques such as frequent in-class problem solving, lab sheets and discussions were used to create an active environment to appeal to a broad range of learning styles. The active learning approach helped students move up from the lower levels of Bloom's taxonomy (simple knowledge and comprehension) into the highest levels (analysis, synthesis, and evaluation). Students attained a greater level of understanding of the material because they had the opportunity to interact with it demonstrated through test scores and student feedback. Students were guided through the development of a hypothesis, testing their particular hypothesis, and explaining how the results support or refute their theory. This process helps students test their knowledge and understanding of a problem.

There is strong evidence in support of the efficacy of these methods. Student outcomes assessment shows that there was an improvement as a result of these techniques. Challenging students to use the scientific method to develop and test hypotheses changed the dynamics of the class, and helped even weaker students actively participate and engage in the material. Standard student evaluation instruments

asked students to identify the most positive aspects of the course. Over 60% of responses listed the in-class labs and class discussions as the most positive aspects of the course.

Active learning techniques can improve the dynamics of a class. The techniques appeal to a variety of preferred learning styles and remove impediments to cognitive processing. Most importantly, we are a publicly funded liberal arts university, and each of these techniques was developed and employed with little cost beyond the class preparation time of the instructor.

Future work will include development of new techniques and application to a wider variety of courses, including Operating Systems and Computer Organization. Active learning creates an environment that helps students share the instructor's enthusiasm for the material and builds confidence in the students.

References

[1] Owen Astrachan. Concrete teaching: hooks and props as instructional technology. In *ITiCSE '98: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, pages 21–24. ACM Press, 1998.

[2] R. M. Felder and R. Brent. Learning by doing. *Chem. Engr. Education*, 37(4):282–283, Fall 2003.

[3] Scott Grissom and Mark J. Van Gorp. A practical approach to integrating active and collaborative learning into the introductory computer science curriculum. In *Proceedings of the seventh annual consortium on Computing in small colleges midwestern conference*, pages 95–100. Consortium for Computing Sciences in Colleges, 2000.

[4] Jeffrey J. McConnell. Active learning and its use in computer science. In *ITiCSE '96: Proceedings of the 1st conference on Integrating technology into computer science education*, pages 52–54. ACM Press, 1996.

[5] K. Silverman R. Felder. Index of learning stylels. World Wide Web., February 2005.

[6] Lynda Thomas, Mark Ratcliffe, John Woodbury, and Emma Jarman. Learning styles and performance in the introductory programming sequence. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 33–37. ACM Press, 2002.

[7] Henry M. Walker. Collaborative learning: a case study for cs1 at grinnell college and austin. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 209–213. ACM Press, 1997.