STATEMENT OF TEACHING PHILOSOPHY

EMILY HILL

Engaged students learn. Regardless of subject, engaged students are more likely to gain a deeper understanding of the material and retain the information after the semester is over. As a teacher and mentor, I try to engage students by involving them in the classroom, using frequent and varied evaluation techniques, working with students individually through research, and by encouraging diversity.

INVOLVING STUDENTS IN THE CLASSROOM

In general, I attempt to involve students in the classroom by increasing class participation, imparting my enthusiasm for computer science , and motivating the material.

Increasing Class Participation

My approach to engaging students in the classroom is inspired by a mathematics professor at my liberal arts college. When deriving a proof or solving a problem, he would ask the class to participate. But rather than wait for the classroom to answer, he would randomly selected an index card containing a student's name. The professor would then work with the student to solve the problem or derive the proof for the rest of the class. When necessary, the professor would prod the student, ask leading questions, or review earlier concepts. Although rather mortifying for students in the beginning, this method of engaging students has a number of benefits:

- Motivates the students to pay attention in class because they never know who will be called upon next
- If the selected student has not been paying attention, by focusing on the student individually, the professor can quickly get the student up to speed and provide a review for the rest of the class
- Students learn from other students' struggles, even if they are uncomfortable asking questions themselves
- Creates an open environment for questions as the students become more comfortable with their peers in the classroom, while ensuring no one student dominates the class
- Ensures the professor does not move faster than the entire class can grasp the concepts

As a student, I will admit that I found this method of learning to be extremely painful, since I was embarrassed to reveal my lack of knowledge in front of my classmates. However, I gradually realized that no student in the class understood everything, and was able to relax and focus on learning the material---something I rarely accomplished during undergraduate lectures.

I can adapt this technique for computer science courses by carefully preparing my lecture material to include focused questions. For example, in a software engineering class I might select students to name challenges in software engineering based on their own programming experience. In an introductory programming or algorithms class, I could select a student to help derive a specific algorithm. Although it requires extra preparation in the beginning, I hope this technique will become a natural part of my teaching style over time.

Imparting Enthusiasm

If the professor is bored, why should the students be excited? When engaging students to learn, it helps to be enthusiastic about your subject. In computer science, I'm particularly passionate about automation. Computers are the quintessential automation tool, capable of infinite variation through software. All the problem solving skills computer science teaches are driven by automation, whether the problem is finding a software bug or developing a more efficient algorithm. As a teacher, I try to infuse my lectures with as much passion as possible, using my presence as a performer to help me communicate my enthusiasm and gauge my audience's response.

Not only am I enthusiastic about the subject, I'm also energized by helping the students understand the material. For example, in lab I noticed students struggling to understand shell scripting and makefiles. To help the students realize the benefits of more advanced command-line software development, I created a shell-scripting and makefile tutorial based on the instructor's example from lecture [see attached tutorial]. In the tutorial I not only explained each line's action, but also how the techniques would reduce their development time.

Motivating the Material

Students are naturally more motivated to learn material that they are interested in or that they see value in beyond graduation. Thus, I also attempt to engage students by motivating the course material, showing how the material is relevant to their future success, and relating the material to their existing knowledge.

For instance, one of the course objectives in an introductory computing class for non-majors is to learn about computer architecture and components. Because I felt the students would most likely use this information in the future to help them purchase a computer, I designed a build-a-computer project [see attached assignment]. Given a budget and a list of components, the students were asked to build a computer that would suit their needs and explain their design decisions based on their own computer use. The students seemed to enjoy both the personalized nature of the assignment as well as its relevance to their future computing needs [see attached feedback].

Furthermore, introductory programming and software engineering courses can be motivated by actual programming and software engineering failures. For example, Verizon's 3G network in Philadelphia was brought down by a missed semicolon, and Washington DC lost cell data service due to an uninitialized variable. More severe examples include the Mars Climate Orbiter crash in 1999 due to ill-defined interfaces (one component used metric units, whereas another used English), or the deaths of cancer patients due to a race condition in the Therac-25 software, which could have been prevented using verification techniques.

Engaging Students through Evaluation

Although engaging students in the classroom is the first step in teaching, frequent and varied evaluation can provide further motivation. Even absent students will generally attempt to learn material on their own in order to maintain their GPA. To keep students motivated to learn each week, I evaluate class participation and give out short weekly quizzes. Weekly quizzes serve two purposes: quizzes encourage students to learn the material as the course progresses, rather than waiting until an exam; and quizzes give a progress report to both the student and professor, suggesting what topics may need to be studied further. For example, when my introductory computing class had trouble with number systems and struggled to convert decimal numbers into binary, I spent extra time going over the material in class and investigated alternate ways to present the lesson.

Exams and projects help motivate students to learn throughout the semester. When possible, I like to assign both individual and group projects that help develop students' analytical problem solving skills and grow their collaborative abilities. Especially for programming and software engineering courses, I try to give programming assignments that start from scratch as well as start from an existing implementation. When starting from scratch, students can exercise their full creativity and see the entire design process from requirements to development and testing. In contrast, starting from an existing code base or requiring the use of a specific library exercises different development skills and more closely mimics current software engineering practices.

Lastly, in my desire to see my students succeed after graduation, I endeavor to include at least one oral or written assignment. For example, in my introductory computing class I assigned a paper about the ethical implications of a computing technology of the student's choice. Not only did the assignment exercise the student's writing skills, it also encouraged them to explore computer science from a different perspective (human rather than technical). In a programming or software engineering course, I could ask students to give an oral presentation about their approach to solving the latest project. Such a presentation would not only encourage critical thinking about design and implementation decisions, but also motivate students to discover innovative solutions to existing problems.

RESEARCH MENTORING: PROVIDING INDIVIDUAL ATTENTION

I believe research mentoring is one of the best ways to motivate students to pursue computer science. With the individualized attention from research and independent studies, students are completely engaged in the research problem. Plus, one-on-one and small group meetings often lead to mentoring opportunities that can provide guidance and motivation to pursuing careers or graduate studies in computer science.

During my dissertation research I had the opportunity to mentor seven undergraduate students. Because a large portion of my work focuses on analyzing code (i.e., reading, not writing code), the nature of my thesis research lends itself to working with students at all levels, from freshmen to graduate students. In fact, I have mentored and successfully collaborated with sophomore, junior, and senior undergraduate researchers. In addition to my thesis research easing student collaboration, my diverse software engineering research background enables me to work with students in a variety of research areas.

Because each student is unique, my mentoring style may vary from person to person. Although I always broadly outline projects for the time allotted, there are some variables with a new student that cannot be anticipated. For example, a student's actual knowledge may be more or less than what it looks like on paper, requiring varying amounts of lead time before the actual research begins. Thus, I try to start off with a short, week-long research assignment to get a feel for the student's skill set and interests before nailing down a specific schedule with deliverables. From my own experience, I know undergraduate researchers sometimes need a push to keep going and maintain a steady level of progress, and I try to set reasonable short term (daily or weekly) and long term (monthly) deadlines, based on the student's skill level and work habits. One of the rewards I look forward to in leading an active research program is observing how others become competent researchers under my leadership and mentoring as we work together on research projects.

ENCOURAGING DIVERSITY

Lastly, I attempt to engage students in computer science by encouraging diversity. I have a history of commitment in attracting people to pursue computer science, especially woman and minorities. As an undergraduate student I initiated and developed two high school outreach programs: one to teach high school students about computer architecture, the other to teach high school girls Flash and HTML basics as well as discuss career opportunities in computing. As a graduate student, I helped co-found CISters, a group that organizes activities to recruit and retain women in technology-driven fields at the University of Delaware. As part of CISters, I co-organized a case study of the status of women in computing at UD, which resulted in travel funding for 10 graduate and undergraduate women to attend Grace Hopper. Finally, I have participated as a co-mentor in the CRA-W/CDC Distributed Research Experiences for Undergraduates program for women and minority students. Of the four undergraduate women and minority students I've mentored, three are either attending or plan to pursue graduate studies in computer science. With both teaching and research activities I hope to continue increasing the number of women and minority students pursuing careers in computer science.

Supplemental Teaching Materials

Emily Hill

TEACHING FEEDBACK

EMILY HILL

QUANTITATIVE FEEDBACK

101 Instructor

Scale: *1 = Excellent, 2 = Good, 3 = Fair, 4 = Poor*

Survey Question	Hill	All 100-level courses
"How would you rate this *course?*"	1.444	2.545
"How would you rate this *Instructor?*"	1.111	2.727

QUALITATIVE FEEDBACK

101 Instructor

"She was very energetic. She spoke and explained everything very simply."

"I really liked the building a computer assignment" [which Emily designed and added to the course]

"The mid-term and final were a little hard"

"I enjoyed the class a lot, especially the teaching styles. I liked how everything was organized."

"I like the way she taught and lectured the class as a group instead of individually."

181 Lab Instructor & Teaching Assistant

"I just want to say Emily is doing great. She returns work at a good pace and won't rill you for making a mistake on a back to back lab (because you didn't receive the other back). She's very willing to help students and never makes you feel stupid. She's willing to try to help you via e-mail and also she is available by appointment. She's doing a great job!"

"Emily is always looking to help however possible; she puts helpful things on her web site for nearly every lab. Now even other TAs are having their students look at Emily's page since it's so helpful."

"Extremely helpful and resourceful, goes out of the way to produce helpful tutorials and teaching elements. Although with grading a little choosy but very understandable why that is."

"Grades a little harsh."

"Emily has helped me greatly during office hours and is a fantastic mentor and resource."

672 Teaching Assistant (graduate level)

From Students:

"The TA's grading and help on the projects was very thorough. She seems very committed and enthusiastic."

"Emily is a good TA. She's easy to talk to when you have questions."

From Professor Pollock:

"Emily was very thorough in her work with lots of comments and extensive testing of labs in 672 this semester. Answered questions in e-mail promptly. She also taught a lecture class for me and did a very good job. Very easy for me to work with!! Made my job easy!!"

CISC 101
Computers & Information Systems
Winter 2004

Labs Homework Handouts Links

Announcements

- If you would like 100 points **Extra Credit**, you may submit by e-mail a 500 word (or more) paper on the History of Computing by 5 pm on Wed Feb 4th. Please site all reference material.
- On **Monday** we covered **Unit 6: Operating Systems**. A Study Guide of concepts covered (to be tested on Quiz 3 & the Final) is available.

Meeting Times:

Lecture	MW	9:45 am - 12 pm	Gore 104
Lab	TR	9:45 am - 1 pm	Smith 040

Instructor
Emily Gibson

Office: 102 Smith Hall
Hours: Mondays 1 pm - 3 pm
E-mail: gibson@cis.udel.edu

TA
Ryan Bickhart

Office: 28 W Delaware Ave
Hours: Tuesdays 8 - 9:30 am
E-mail: bickhart@cis.udel.edu

Schedule
**Updated 1/25/04*

	Lecture	*Lab*	*Lecture*	*Lab*
Week	**M**	**T**	**W**	**R**
1/5		1-2	Assign Paper	3
1/12	Quiz 1 Assign HW1	4	Quiz 2 (Number Systems, 4)	12
1/19	Martin Luther King Day No Class	9	HW1 Due Assign HW2 Mid Term Paper Topic Due	10
1/26	Unit 6	11	Quiz 3 HW2 Due	5
2/2	Quiz 4 Paper Due	6	Final	8

Syllabus **(HTML) (WORD)**

Labs

1. Lab 1 WWW
2. Lab 2 Intro User Interfaces & E-Mail
3. Lab 3 Word Processing Software: WORD
4. Lab 4 Word Processing Software: WORD
5. Lab 12 Presentation Software: PowerPoint
6. Lab 9 Spreadsheet Software: Excel
7. Lab 10 Spreadsheet Software: Excel

8. Lab 11 Spreadsheet Software: Excel
9. Lab 5 Database Management : Access
10. Lab 6 Querying a Database: Access
11. Lab 8 HTML

Lab download and print instructions

Homework

- HW1: Number Systems (HTML) (WORD)
- HW2: Computer Systems (HTML) (WORD) (EXCEL)
- Ethics Topic Paper (HTML) (WORD)

Handouts

- Unit 6: Operating Systems Study Guide (HTML)
- Dye Sublimation Printers (HTML) (WORD)

Links

- **Binary/Hex Text Converter**
- **Next Generation DVDs** (2005)
  HD DVDs (tech by Toshiba) can hold 15 GB of data with a single layer, or 30 GB with dual layers! However, as with the DVD recording technologies, the higher capacity DVD technology debate is still raging.
- **Dye Sub Printers Explained**
- **Number Systems**
  - Explanation of number systems and how to convert between them.
  - More on converting between decimal, binary, and hex.
- **CIS Homepage**
  - Courses for Non-Majors
  - CISC 101 Course Description
- **UD Homepage**

HW2: Building a Computer System

Description
Imagine you have an $800 budget to build a new computer for yourself, to *replace* the one you are currently using. List the components you would use, based on the list from the class web page. Make sure to explain *why* you are selecting each part. In addition, specify why you are *not* including items. (For example, if your Motherboard has an onboard LAN adapter, state that you don't need to buy an Ethernet card for this reason).

Feel free to use other web sites if the product links provided do not give you enough information. This is a challenging assignment, if you don't understand something, please ask.

Things to consider when building your machine:

- **Price**
  Maybe you would prefer to save the extra $300 and only spend $500 on your machine.
- **Performance**
  How much computing power do you need?
- **Peripherals**
  Would you rather have more gadgets than a bleeding edge PC?
- **Expandability & ability to upgrade**
  Can you upgrade this machine later?
- **Loudness**
  If the PC is going to be in your bedroom, you may want to consider quieter parts.

Grading
The components you choose and reasons for (or not) selecting them are worth 75 points. In addition, you must write an explanation detailing your design considerations (the overall design you had in mind when building the PC). This explanation is worth 25 points.

Product	**Price**
Case	
Mercury Primo ATX PC Case with 300Watt Power Supply	$30.00
Soyo - Black and Silver Mid-Tower Case with 350 Watt Power Supply, Keyboard, Mouse, and Speakers	$40.00
Lian-Li PC 6070A Aluminum Ultra Quiet	$169.00
Shuttle SB51G Intel 845GE 400/533 FSB (Barebones - Case w/Mainboard)	$225.00
Power Supply	
Data Pro ATX 300 Watt P4/AMD Power Supply with Fan	$22.00
400 WATT PS (Dual FANS) AMD-INTEL READY	$39.00
NX-3500 Nexus 350 Watt Special Edition Quiet Power Supply	$80.00

Shell Scripting

- **What is a shell script?**

A shell script is a (text) file with shell commands in it. What's a shell command? It's any command you can type at the command prompt, for example, when you are logged in to strauss or copland.

- **What do I use a shell script for?**

Do you find yourself typing the same few commands over and over and over again? (For example, for lab05b you typed the same commands over and over: CC, a.out, gnuplot, cp, chmod.) If you created a script `mydrawing.csh`, then instead of typing 5 commands, you could type one:

```
mydrawing.csh
```

And have all 5 commands execute automatically!

- **How do shell scripts work?**

The following is an example shell script file from lecture. Click on each line for an explanation of the command.

```
#!/bin/csh

clear
date
g++ -c main.cpp
g++ -c drawings.cpp
g++ -o main main.o drawings.o
main
gnuplot mydrawing.gnuplot
cp mydrawing.png ~/public_html/cisc181/lab06
chmod -R a+rx ~/public_html/cisc181/lab06
```

Makefiles

- **What is a Makefile? Why would I use one?**

A Makefile is a (text) file with make commands in it. What's a make command? A make command is a make rule that looks like this:

```
lab05b: lab05b.cc
        CC -o lab05b lab05b.cc
```

If I type `make lab05b` at the command line (assuming I have a file called Makefile in my directory with the above lab05b rule in it), then the line

```
CC -o lab05b lab05b.cc
```

will be executed. So why not just type in the `CC` command yourself? The beauty of make is that it will execute the `CC` command *only* if your lab05b.cc file has changed! So unlike the script file above (which compiles main.cpp and drawing.cpp regardless of whether or not the files have been modified), make is "smart": it only compiles what it needs to.

Let's break down the make rule:

```
target: prerequisites/dependencies
        command
```

The target is what you type at the command line (`make target`) to execute a rule. Following the colon on the first line you list all the files or other targets that the current rule is dependent on. Basically, if any of the files listed on the first line have been modified, we want make to execute our command. Commands for a rule are listed below the first line, **and begin with a tab** (one and only one). You may have multiple commands for a single rule.

Makefiles may not seem important now, but if you continue to develop programs, I promise you will learn to love them. I used to use graphical IDEs (Integrated Development Environments) such as Microsoft Visual Studio. But once I learned how to effectively use shell scripts and make, I switched how I develop programs and never looked back. Remember, programmers used shell scripts and make 20 years before Microsoft Windows was even invented!

- **How does make work?**

The following is an example Makefile inspired by the above script. A more complex example with comments was discussed in lecture. Click on each target for an explanation of the rule.

```
# Example Makefile

main.o: main.cpp
        g++ -c main.cpp

drawings.o: drawings.cpp
        g++ -c drawings.cpp

main: main.o drawings.o
        g++ -o main main.o drawings.o

execute: main
        clear
        date
        main
        gnuplot mydrawing.gnuplot
        cp mydrawing.png ~/public_html/cisc181/lab06
        chmod -R a+rx ~/public_html/cisc181/lab06

clean:
        rm *.o


# Next step: add a variable for the compiler

CCC=g++

main.o: main.cpp
        ${CCC} -c main.cpp

drawings.o:     drawings.cpp
        ${CCC} -c drawings.cpp

main: main.o drawings.o
        ${CCC} -o main main.o drawings.o
```

`mydrawing.csh`

`#!/bin/csh`
There are a number of different shells available, such as csh (c shell), bash, sh, tcsh, etc. When you execute a script file, the first line (prefaced by #!) dictates what shell language will be used. The languages are very close, but may differ in syntax.

`clear`
The clear command clears the terminal window, so you start from a "cleared" terminal screen.

`date`
Date displays the day and time.

`g++ -c main.cpp`
Previously, when we compiled using CC or g++, we typed `g++ lab.cc` and an a.out file, or an executable file, was created. The compiler was hiding a step from us: linking. When you use more than one C++ file, you must compile each file seperately and then link them all together to form the executable.

Why use seperate files? First, it might be convenient to seperate different functions into different files. This way, you can reuse your drawing functions with a different main program, like main2.cpp, for example. Also, if you change your main.cpp file you don't need to recompile your drawing.cpp file, or vice versa.

The command above compiles the main program, main.cpp, which calls functions from drawing.cpp (see below). The -c option tells g++ to create main.o (the object file) and to skip the linking step (the step that usually creates a.out).

`g++ -c drawings.cpp`
The command compiles the drawing functions, drawings.cpp, which are called from main (see above). The -c option tells g++ to create drawings.o (the object file) and to skip the linking step (the step that usually creates a.out).

`g++ -o main main.o drawings.o`
The linking step. Now that we've compiled drawings.cpp and main.cpp, we need to create the executable file. The -o option tells the compiler to name the executable main instead of a.out. We list the object files main.o and drawings.o because we need the code from both of them in our executable (main.cpp calls functions from drawings.cpp).

`main`
Execute our program.

`gnuplot mydrawing.gnuplot`
Execute gnuplot, using the mydrawing.gnuplot file.

`cp mydrawing.png ~/public_html/cisc181/lab06`
Copy the png image created by gnuplot into our web directory.

`chmod -R a+rx ~/public_html/cisc181/lab06`
Make sure our png file has the correct permissions (so we can actually **see** our graphic!).

Makefile

```
# Example Makefile
```

As with shell scripts, a # sign begins a comment.

```
main.o: main.cpp
        g++ -c main.cpp
```

The main.o target is dependent on the main.cpp file. If main.cpp has not been modified since the last time main.o was created, don't execute the command. If main.cpp has been modified or main.o doesn't exist yet, run `g++ -c main.cpp` to create main.o.

```
drawings.o: drawings.cpp
        g++ -c drawings.cpp
```

Similar to the above rule, only this rule is applied to drawings.o and drawings.cpp.

```
main: main.o drawings.o
        g++ -o main main.o drawings.o
```

The linking step. (Be sure you have read and understood the compiling and linking explanations from the shell script.) Creation of the executable file depends on the object files of main.o and drawings.o. (Notice how the command `g++ -o main main.o drawings.o` uses both these files to create the main executable.)

Here comes the cool part:
By typing `make main`, the rules for main.o and drawing.o will automatically be executed! So by typing one command, make checks to see if main.o needs to be re-compiled, checks to see if drawings.o needs to be recompiled, and then links these two object files together to create our executable, main! If compilation fails (i.e. there's an error) for either main.cpp or drawings.cpp, make will halt and will **not** execute the `g++ -o main main.o drawings.o` command.

```
execute: main
        clear
        date
        main
        gnuplot mydrawing.gnuplot
        cp mydrawing.png ~/public_html/cisc181/lab06
        chmod -R a+rx ~/public_html/cisc181/lab06
```

Make rules are not just for compiling programs, you can create rules with any type of shell command. In lab05b, for example, we needed to execute 5 or more commands just to see if our C++ program produced correct results (in the form of a png image file that could only be viewed on the web). Typing `make lab05b` doesn't save any typing, it only keeps us from recompiling unnecessarily.

Instead, we could create a make rule that is dependent on our executable, and executes all the commands we need *only if creation of the main executable is succesful*. By typing `make execute`, make will first execute the main target. Remember, the main target compiles main.cpp and drawings.cpp if necessary and then links the object files together to form the main executable. If the main rule successfuly creates the main executable, then the execute rule will run the following commands in order:

1. `clear`
2. `date`
3. `main`
4. `gnuplot mydrawing.gnuplot`
5. `cp mydrawing.png ~/public_html/cisc181/lab06`
6. `chmod -R a+rx ~/public_html/cisc181/lab06`

Which will automatically take our .dat file (output by running main), run gnuplot, copy the png image to our web site, and then make sure our image is readable, *all with one command!* Are you hooked yet?

```
clean:
        rm *.o
```

Once we've successfully created an executable, we don't need the temporary object files that were created before the linking step. This rule just cleans up our directory and gets rid of clutter.

```
# Next step: add a variable for the compiler
```

Moving on to a new Makefile...

```
CCC=g++
```

Like programming, variables can save you typing. In make variables can be used for which compiler you're currently using. For example, by creating a compiler variable CCC, if I ever want to change the compiler my make rules use from g++ to CC, all I have to change is the CCC variable!

```
main.o: main.cpp
        ${CCC} -c main.cpp
```

Same main.o rule as above, only now we use the compiler variable CCC instead of g++. Notice that to use our variable we put the variable name inside ${}.

```
drawings.o:     drawings.cpp
        ${CCC} -c drawings.cpp
```

Similar to the above rule, only this rule is applied to drawings.o and drawings.cpp.

```
main: main.o drawings.o
        ${CCC} -o main main.o drawings.o
```

Similar to the above rule, only this rule for the linking step.