Beyond Objects

Lynn Andrea Stein*
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139
las@ai.mit.edu
(617) 253-2663
fax: (617) 253-5060
http://www.ai.mit.edu/people/las
http://www.ai.mit.edu/projects/cs101

Abstract

Many schools are considering a switch to Java in their first course, "CS1". If we do this, we can continue to teach approximately our existing courses, sweeping a few nasty details (e.g., the enforced concurrency of AWT) under the rug. Alternately, we can use this switch to make a fundamental — and perhaps long overdue — change to the way that we teach introductory programming.

This past fall at MIT, I taught a rather unusual introductory programming course. It used Java, but it was not much like a traditional introductory programming course. Its main focus was on the idea of COMPUTATION AS INTERACTION. In this view, programming is about constituting a community of interacting processes, and the fundamental questions one asks have to do what goes inside each of these community members and what goes between them. This paper outlines my course.

1 What we teach

Typically, an introductory computer science course teaches students how to write computer programs. The aim of this approach is twofold: on the one hand, students learn an important skill, a tool that will serve them well within this discipline. The more fundamental reason why so many schools teach programming first, however, is that programming teaches students

*This material is based upon work supported by the National Science Foundation under a National Science Foundation Graduate Fellowship and under National Science Foundation Young Investigator Award Grant No. IRI–9357761 to Professor Lynn Andrea Stein. Any opinions, findings, 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. This work is also supported by the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology and by MIT's Class of 1951 Fund for Excellence in Education and Class of 1955 Fund for Excellence in Teaching.

the basic vocabulary of computer science, including what computation is, how it works, and what questions one can ask about it.

Introductory computer science is where students learn the basic vocabulary of computation, the fundamental language that we as a field use. Over the decades, this language has shifted, first to include procedures, more recently to incorporate objects. In this paper, I describe a transition that we have not yet made, but that I see as crucial to the way we as practitioners think about computation. I describe an introductory computer programming course centered around the idea of computation as interaction and explain why I believe that this approach provides a more appropriate vocabulary for today's computer science students.

When computation was first performed mechanically, its basic vocabulary was the sequencing of primitive steps. The addition of procedural abstraction allowed a more abstract conceptualization of computation, roughly in terms of input-output functionality. Rather than manipulating data in single primitive steps, programs with procedural abstraction allow a collection of these steps to be treated as a new primitive. The basic vocabulary of this kind of computation is that of functional behavior.[1] Much of computational theory (as well as, e.g., algorithm design or even software engineering) was developed around this model of computation.

More recently, the notion of programming in terms of objects has taken a central place in our computational vocabulary. Objects encapsulate functions together with state. This means that a call to an object's method may be important not for its (functional) behavior, but for its effect on that object's subsequent behavior. Entering a new phone number into my rolodex doesn't do much for me, but later it means that the rolodex can supply me with that information. Objects are the first half of the shift that I wish to describe in this paper.

The second half of this transition has to do with autonomy. In most traditional object-oriented programming languages, objects are repositories of behavior. In a few, objects actually take on lives of their own as independent actors. This changes the fundamental vocabulary of computation further: not only are functions replaced by interaction patterns, but these interaction patterns take place concurrently and asynchronously. Programming no longer (necessarily) involves designing THE flow of control in a system; instead, it is fundamentally about constituting a community of autonomously interacting entities, deciding what goes inside and what goes between each.

In this paper, I describe a course which, like many introductory programming courses, teaches its students to think in terms of the fundamental vocabulary of computation. The novelty of my approach is in what I take to be that fundamental vocabulary. First, the atomic unit of computation is NOT print("Hello, world!") — the paradigmatic "step" or function of computation. Instead, the atomic unit of computation is

```
while(true){
   echo();
}
```

— an autonomous interactive control loop which constantly monitors and responds to its environment. Second, the questions this leads one to ask about computation are different:

[1] Of course, there have always been "side effects"; these can be incorporated by thinking of the state that they "touch" as a part of the input and/or output domain(s).

not "What value does it produce?" or "How long does it take?", but "What are the entities in constituting the computational community?", "What resources does each control?", and "How do the members of this community interact?"

In the remainder of this paper, I sketch an outline of such a course. The course was taught on an experimental basis at the Massachusetts Institute of Technology in the Fall of 1996. The students enrolled were (largely) first term freshmen; no prior programming experience was assumed.

2 Unit 1: Language basics

The first unit of this course covers the basics of the Java programming language. Although the course could in principal be taught in other programming languages, at present there is no language better suited to its needs.

In this unit, students learn to write Java program files. These are neither applets nor standalone Java programs; instead, they run in conjunction with a wrapper program that we supply. This means that, from the beginning, students are building complete units; also from the beginning, they are building units that interact. By the end of this unit, students have seen instances and classes, methods and fields, and the basics of Java expression and statement construction. In these ways, this unit is a traditional introduction to the vocabulary of object-oriented programming.

This introduction to Java is unusual on several points. First, control flow and complex statements are not covered at this point in the course. Second, there is less emphasis on inheritance, and in particular on building up complex hierarchies, than in a traditional OOP approach. Although covered later in the course, extended inheritance examples are less central to the autonomous-object approach. Finally, Threads (Java's concurrent execution primitives) are introduced almost at the beginning of the course. In this course Threads are always used in a particular stereotyped way. The ActiveObject class (Figure 1) demonstrates this use of Threads: always created and invoked on this, never extended, and generally used with a run method that contains a while(true) loop.

By the end of this unit, students have learned to construct active autonomous objects:

3 Unit 2: Systems of interactions

In the second unit of this course, the students are introduced to systems of interacting objects. The goal of this unit is for students to observe, predict, and orchestrate behavior of and in these systems. Two example programs illustrate this point.

One of the first programs that students build is a player in a simple two-player balancing game. A (roughly) horizontal rod appears on the screen. The student's code controls the vertical position of one end of the rod. Another process asynchronously controls the other end. Students write code to keep the rod balanced. Initially, they test their code against their code: two instances of their class control the two ends. Once they have this working, they test on instructor-supplied placebos and on one another's code.

This exercise teaches students to build simple autonomous interactive control loops, but also to integrate them into a system. They learn about assumptions — their code should

```
public abstract class ActiveObject implements Runnable {

     private protected Thread spirit;

     public ActiveObject(){
         this.spirit = new Thread( this );
         this.spirit.start();
     }

     public void run(){
         while( true ){
             this.act();
         }
     }

     public void act();

}

class Echo extends ActiveObject {

     void act(){
         System.out.println( System.in.read() );
     }

}
```

Figure 1: The `ActiveObject` class is a fundamental unit of computation.

be indifferent to which end of the rod it sits on — and about interactions — their code's behavior is determined by the combination of what they wrote and what their code interacts with. They also have an opportunity to collaborate and/or to compete.

A later exercise in this unit is a simplified network router. This simulation includes a graphical visualization of a network of nodes. We provide the source and terminal nodes; they provide the behavior of the intermediate nodes. Nodes are expected to move packets from their input streams to their output streams without creating or losing packets. The visualization allows students to configure their own test networks and to dynamically modify their networks — bringing up new connections or causing existing connections to fail — while the simulation is running. This exercise stresses the notion of error handling, including Java's exception-handling structures, and the idea of robustness to failure. It also introduces the array data type and conditionals. More importantly, the visualization allows students to observe the implications of their programs interacting in context and as a part of a larger system. Among their tasks are the design of test networks, the testing of their behavior(s) on others' networks, and the prediction of both successful and anomalous behavior.

At the end of this unit, students have a better sense of the ways in which behavior can be coupled into group dynamics and the effects of varying component behavior on system behavior.

4 Unit 3: Inside: Control Flow

In the next unit of the course, we turn to the behavior of individual active objects. Many of the main points of this unit can be illustrated here with a single example (though in the class several are used to reinforce the major points).

A simple four-function calculator provides an easy-to-understand user interface. Students are expected to write the back end "brains" of the calculator. This is an excuse (initially) to introduce the notions of procedural abstraction and (control structures for) dispatch: handling the 1 button is much like handling the 3 button, and there are language mechanisms to facilitate this. The first calculator is thus an example of an interactive control loop whose main behavior is dispatching (based on button type) to encapsulated behavior for handling each type of button.

But dispatch need not be the central idea here. Instead, we can shift the emphasis away from the central control loop and into the dispatched-to procedures. This is essentially the idea behind event-driven programming: a behind-the-scenes implicit `while(true)` loop is dispatching on event type to the appropriate handler object. The idea of event-handling as a programming paradigm is illustrated here by introducing the Java AWT user interface toolkit.

In addition to introducing control structures for dispatch (e.g., switch), procedural abstraction, and event-driven programming, the calculator provides an opportunity to teach students a bit about formal modelling tools. The simplest behavior of a calculator is reasonably modelled using a finite state machine, though more complex versions are easier using an augmented finite state machine. Alternately, some of the object-oriented modelling tools can be used. In any case, the calculator object needs to keep track of certain aspects of its state, introducing students to a more traditionally object-oriented view of programming.

There is one more important point to be made in this unit: the use of polymorphism as a way of implementing an implicit dispatch. In this version of the calculator, each button is itself implemented as a "smart object" capable of "handling" itself. The central control loop of the calculator is now simplified from a large switch statement to a one-line control loop that delegates state updates to the "button brain" of the GUI object just pressed. This approach, too, exploits more of traditional object-oriented software engineering and stresses concepts such as inheritance.

5 Unit 4: Between:

The final unit of the course involves building communities whose constituents are themselves communities. An example from this section is a networked cat-and-mouse chase game.[2] Students work in pairs, one implementing the cat, the other the mouse. Each side of the application is itself a community of interacting processes: at least the AWT thread and the network communication thread, and potentially others as well.

Each student must therefore constitute a community of interacting entities — the UI handler, the network handler, etc. Building this component successfully involves a basic understanding of shared state and synchronization. The design space here is limited enough for students to explore safely, but complex enough for them to learn something from this exploration.

In addition to building their individual components, each pair of students must agree on a particular interface between their components. Because the interface is not fully specified — where, e.g., do the position comparisons happen? — there are many satisfactory solutions. The differences in implementation provide the basis for significant discussion. Solutions include "server push", "client pull", and the use of a "repository" (blackboard?) object.

From this exercise, students take away several lessons. First, there are many right solutions. Second, no solution can hope to succeed unless both (all) sides agree on a shared interface. One group's cat will likely not run properly with another group's mouse. Third, this exercise is a jumping-off point for a more thorough discussion of distributed systems architectures. By considering a many-player version of this game, students can easily understand such issues as centralized server bottlenecks and the (potential) benefits of peer-to-peer communication. Finally, the exercise illustrates the recursive application of the building-a-community metaphor. From the perspective of the whole cat, the whole mouse is a monolithic entity; within the mouse, finer detail is evident.

6 Discussion

In this paper, I have described what I see as the next step in introductory computer science education. Computation today is largely interactive: concurrent, distributed, ongoing, condition-monitoring, event-driven, and not computing a particular answer. (In many of

[2] Each player sits at a computer. One player — the mouse — moves around the screen. The other — the cat — watches the mouse (whose position has been transmitted across the network) and tries to "catch" it by clicking on it. The mouse may also see the cat to better be able to escape it.

today's programs, stopping means something's gone wrong.) While most of us practice computation of this nature — video games and spreadsheets, databases and operating systems, controllers and web servers — few of us preach it to our students.

The course I have described (briefly) here is one that has been taught to first-term undergraduates with little or no prior computational experience. (By the time of OOPSLA, it should have been taught over the summer and begun in the fall on multiple campuses. Further details are available on the web at `http://www.ai.mit.edu/projects/cs101`) According to the student-written course evaluation, "The class is a good introduction to many important principles of computer science." These principles amount to the basic vocabulary of a new kind of computation, one which is the natural next step beyond objects.