Follow the River and You Will Find the C

Jae Woo Lee
Dept. of Computer Science
Columbia University
New York, NY, USA
jae@cs.columbia.edu

Michael S. Kester
Dept. of Computer Science
Columbia University
New York, NY, USA
msk2117@columbia.edu

Henning Schulzrinne
Dept. of Computer Science
Columbia University
New York, NY, USA
hgs@cs.columbia.edu

ABSTRACT

We present a one-semester transition course intended to bridge the gap between a Java-based introductory sequence and advanced systems courses. We chose to structure our course as a series of lab assignments that, while independent, are also milestones in a single main project, writing a web server from scratch. By anchoring the course on a single real-world application, we were able to provide depth, instill good programming practices, give insight into systems, and generate excitement.

Categories and Subject Descriptors

K.3.2 [**COMPUTERS AND EDUCATION**]: Computer and Information Science Education—*Curriculum, Computer science education, Information systems education*

General Terms

Design, Human Factors, Languages

1. INTRODUCTION

Current trends in the introductory computer science sequence follow the objects-first paradigm. Java is the language of choice in most cases [10], while some have argued Python as a good alternative [9]. Others have resisted the trends and remain committed to a traditional imperative-first or even functional-first approach. The subject remains an ongoing debate [11, 7, 8, 17] and the authors take no position in this paper. Rather, we address a concrete problem that arises as a consequence of choosing objects-first.

Our school has adopted the objects-first approach. Specifically, it follows the three course sequence recommended by CC2001 [6]. This conversion, however, created a gap in knowledge as the students progress to upper level courses like operating systems and computer graphics, where they need a command of C and the UNIX environment.

This "gap problem" seems common among our peer schools. The nationwide decline in enrollment has forced educators to seek ways to attract and retain students. This has led to many changes in the introductory sequence the most prominent of which is the shift to the objects-first paradigm [13]. On the other hand, the upper level courses have largely remained the same, in part because retention of students is less of a concern, and in part because the focus is on the current body of knowledge rather than pedagogy. Most schools that opt for objects-first offer a transition course aimed at bridging the gap. A review of the curricula of our peer schools showed that transition courses typically attempt to cover too many topics. This either limits the course to a cursory treatment of the material, which may leave the students underprepared for advanced coursework, or results in a course where success requires substantial prior programing experience. Such a course would alienate many promising future CS majors.

This paper presents a C-based introductory systems programming course [1] designed to avoid common problems associated with transition courses. We explore the principles of our course design in Section 2. We present a detailed layout of our course in Section 3. We discuss survey results and student experiences in Section 4. Lastly, we summarize our contribution in Section 5.

2. DESIGN PRINCIPLES

In our search for a suitable model for our transition course, we found surprisingly little research about structuring such a course. We feel this is largely based on two factors. First, a typical introduction to UNIX and C is probably considered fairly bland from an educational research standpoint. On a more practical level, the shift to objects-first is a relatively new phenomenon, so educators have yet to come to a consensus on the effect it has on the CS curriculum.

Our course is designed as a single project with milestones, each of which we call a *lab*. Each lab constitutes a step toward completion of the project without explicitly referring to it as such. Students are given one to two weeks to complete each lab assignment, and are expected to work independently. Shortly after each due date a fully commented solution is provided. This course overlaps with the last semester of the introductory sequence, which is, introduction to programming (CS1), object-oriented design (CS1.5), and data structures and algorithms (CS2), each taught in Java. Our school recommends our course be taken concurrently with CS2.

For the single-project approach to be most effective, it is critical to achieve four sub-goals. (*a*)*Don't forgo depth.* Upon completion of this course, the students should be ready

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

for Linux kernel programming in our school's operating system course [20], which requires a strong C and UNIX background. (*b*)*Focus on doing it right.* We emphasize writing bug-free code, rather than simply producing the required results. (*c*)*Lay out the big picture.* At the end of the course the students should have a clear concept of how a complex software system works. (*d*)*Don't be boring.* The students should experience the joy of programming.

We chose writing a web server from scratch as our big project. Each lab, though self-contained, contributes an individual part of the system, either with subsystem code or a fundamental concept. This structure also affords us a surprising degree of breadth as we incorporate concepts from data structures, databases, operating systems, networking, and multi-tier server architectures. This is similar in spirit to "teaching breadth-first depth-first" as put forth by Murtagh [19, 18]. He advocates, rather than teaching the normal wide ranging topics found in a CS1 course as a disjoint set, exploring them as a whole through the lens of a single subfield, computer networking.

Building a web server from scratch is far more effective in motivating students than completing programming exercises. Literature supports that students respond far more enthusiastically to working with production software than pedagogic software [14]. Our students reported exultation when they saw a web page appear in Firefox, served by code they wrote (achieving sub-goal *d*). The *surprise* they experience from building something large and real at this early stage has a lasting impact on their education [21].

We address the remaining sub-goals in a similar fashion. The web server was required to communicate with a back-end server and generate pages dynamically. As such, students thoroughly learn how web-based multi-tier software architectures work (*c*). Coding a web server requires a strong command of C (*a*). We put an emphasis on proper memory management (*b*). Memory errors are a primary source of bugs for novice C programmers [16].

3. THE COURSE

In this section, we describe the course implementation. There are ten lab assignments. The completed web server is lab 7. In a way, the course forms a narrative following a typical structure of a drama: an exposition (labs 1-3) to lay the groundwork of what is to come, a rising action (labs 4-6) to bring the participants ready for the climax (lab 7), a falling action (lab 8) to wind down the climax, and a resolution (labs 9-10) to close the course.

3.1 Exposition

Labs 1-3 introduce the basic tools and skills needed in later labs. At the completion of this section of the course, students will be familiar with the most common UNIX commands, be able to use pointers, and understand the linked list data structure.

Lab 1: UNIX Basics, SVN, and Make

"The beginning is the most important part of the work."
- Plato

Lab 1 provides an introduction to three components critical to the remainder of the course: UNIX, SVN, and Make. We help the students set up their shell environments, and introduce them to the UNIX man pages, encouraging them to learn new commands as they encounter them. We provide a short tutorial on SVN, which the students are required to use throughout the semester. In the first semester we taught the class, the students were expected to learn Make on their own from the online documentation, which proved to be insufficient as a tutorial. In subsequent semesters, we provided a tutorial during lecture. In both cases, we provided a heavily commented solution for future use as a template.

The deliverable for lab 1 was a trivial program. The goal was simply to introduce the tools that would become the buttresses of their C programming repertoire.

```
int main(int argc, char **argv)
{
    if (argc <= 1) return 1;
    char **copy = duplicateArgs(argc, argv);
    char **p = copy;

    argv++;
    p++;
    while (*argv) {
        printf("%s %s\n", *argv++, *p++);
    }
    freeDuplicatedArgs(copy);
    return 0;
}
```

Figure 1: The main function provided in lab 2.

Lab 2: Pointers and Arrays

"It can be hard to teach them in half an hour, but we can give them a few pointers. . . " - Scott Morrison

Lab 2 builds a solid understanding of pointers and arrays in C. For example, the most challenging problem in lab 2 asks the students to properly create and destroy a copy of the `argv` structure. The students are given the main function of a program called `twecho`, shown in Figure 1, which echoes each command line argument twice, once verbatim and once capitalized. The students are asked to implement the two functions it calls: `duplicateArgs` and `freeDuplicatedArgs`. The former duplicates the `argv` structure into a newly allocated structure and the latter frees the structure. Writing the two required functions, without making any memory errors, requires a solid understanding of how to allocate and destroy a non-trivial memory structure. It also requires an understanding of how to manipulate pointers to access and modify data in memory locations.

In this lab we introduce Valgrind [5]. Valgrind is a error checking utility that detects memory leaks, overrunning array bounds, and a number of other common errors. Valgrind enables the identification of memory errors in a program even when the program appears to work. This is consistent with our *focus on doing it right.* Lab submissions that produced the correct results at the expense of memory errors were heavily penalized.

The initial foray into memory management tends to be challenging for beginning C students. As such, we allowed ample time for this assignment, the teaching staff provided significant support to help the students during office hours, and a class mailing list provided a forum in which the students could discuss common problems. Our goal was not only to help them get their code working, but also to make them understand it.

```
struct Node {
    struct Node *next;
    void *data;
};
struct List {
    struct Node *head;
};
struct Node *addFront(struct List *lst, void *data);
struct Node *findNode(struct List *lst, const void *dataSought,
    int (*compar)(const void *, const void *));
```

Figure 2: An excerpt from lab 3's header file.

Lab 3: Implementing a Linked List

"In all planing you make a list and you set priorities."
- Alan Lakein

Lab 3 introduces the students to data structures in C by having them implement a singly linked list. While this is a typical assignment in a C programming course, we present it in a manner consistent with our prevailing principles. We specify the prototypes, some of which are shown in Figure 2, and we provide a comprehensive test driver program. The students are required to run the test driver using Valgrind in order to make sure that their implementation is free of memory errors.

We also introduce two new concepts: callback functions and libraries. Several linked list functions rely on a pointer to function, such as the `findNode` function shown in Figure 2. The students follow a tutorial to build a library from their linked list implementation, which they are required to use in subsequent labs.

The linked list follows pointer semantics. Each node holds a `void*` to a generic data item, but does not manage the lifetime of that item. The interface of the linked list is revisited later in lab 10, which asks the students to wrap the pointer-based linked list in a new value-based interface using C++.

3.2 Rising Action

Labs 4-6 focus on details that are essential to the ultimate goal of writing a web server. The topics include I/O, process management in UNIX, TCP/IP, Sockets, and HTTP.

Lab 4: Standard Input and Output

"A library is thought in cold storage."
- Herbert Samuel

The lectures at this point cover I/O in C. Topics include standard input and output streams, redirection and pipes in UNIX, formatted I/O functions, and file I/O. These concepts are exercised in lab 4. The students must implement *MdbLookup*, a program that searches through a message database. The message database is a binary file in which each record has a simple structure of two fields, name and message which are limited to 15 and 23 characters respectively:

```
struct MdbRec {
    char name[16];
    char  msg[24];
};
```

When the program starts, every message of the database is read into memory using the linked list from lab 3. The program then enters a loop prompting the user for a search string and matching it with the records. An empty string, which can be input by simply pressing *Enter* at the prompt, matches all records.

Lab 5: Turning MdbLookup into a server

"The old pipe gives the sweetest smoke."
- Irish Proverb

At this point we have concluded our discussion of C, and shift focus to UNIX and TCP/IP networking. During lecture, we give a brief overview of operating systems and TCP/IP networking, without going into detail. The main focus is to convey to the students the concept of "layers." We first explain the layers in software systems: OS kernel, system calls, library functions, and applications. We then introduce the five layers of the Internet protocol stack, again with limited detail. The sockets API is an interface between layers 4 and 5, which allows us to treat layers 1 through 4 as a black box. The students had no problem with these concepts because of their daily experiences with operating systems and the Internet. We also introduce the concept of processes, and how a new process in UNIX is created using `fork` and `exec`.

Here we introduce Netcat [2], a command line program that creates a TCP socket and connects it to the standard I/O. The incoming end of the TCP connection is connected to `stdout` and the outgoing end of the TCP connection is connected to `stdin`. It can run as either a client where it will initiate a connection, or server where it will listen on a port. The client mode operation is similar to that of Telnet.

Lab 5 demonstrates how to combine the MdbLookup program and Netcat in a pipeline forming a network server without sockets programming. We provide the pipeline and the command to make a named pipe:

```
mkfifo mypipe
cat mypipe | nc -l -p 40000 | mdb-lookup > mypipe
```

The standard input and output of MdbLookup is connected to those of a Netcat session using pipes. Netcat in turn connects its standard input and output streams to a socket connection. The students are asked to wrap the pipeline in a shell script and control the script from a parent program which calls `fork` and `exec`.

One of the goals here is to convey to the students the central tenet of UNIX: small is beautiful [12]. A complex task is broken up into small building blocks, and the standard I/O paradigm makes it possible to connect them together.

Lab 6: Sockets Programming and HTTP

"From now on, I'll connect the dots my own way."
- Bill Watterson

Lab 6 covers two topics that are needed to implement a web server: sockets programming and HTTP. During lecture, we walk through simple TCP client and server examples. The server echoes back strings sent from the client.

The first part of lab 6 asks the students to write a version of MdbLookup server using sockets. There is actually very little code to write. They combine the lab 4 program and the TCP echo server example to make the MdbLookup server. The solution we supplied merged the two programs and required fewer than 20 lines of modifications.

The second part of lab 6 asks the students to write a limited version of Wget [3], a command line program that downloads a given URL. The students were only required to implement basic single file download functionality. In preparation for this part, we demonstrated how HTTP works by capturing the dialog between a web browser and server.

First, we use Netcat in client mode to pose as a web browser. We connect to a real website and take note of the headers generated by the server. Next we use Netcat in server mode to see what headers the browser had sent.

3.3 Climax

Lab 7: Writing a Web Server

"If we fix a goal and work towards it, then we are never just passing time." - Anna Neagle

Lab 7 is the culmination of all efforts up to this point. It is a serious undertaking, but the students have all the necessary tools and background. The assignment is to code a web server that implements a subset of HTTP 1.0. The server handles GET requests for static content. The server does not send a `content-type` header, but modern browsers can usually detect the types of the incoming files, enabling the retrieval of HTML pages with images.

In addition to static pages, the server is required to return dynamic content, specifically MdbLookup records. The web server provides an HTML front-end to the MdbLookup server from lab 6. The web server maintains a persistent TCP connection to the MdbLookup server. GET requests in the form of `/mdb-lookup?key=`*`string`* trigger the web server to query the MdbLookup server. The web server formats the returned messages into an HTML table.

3.4 Falling Action

Lab 8: Writing an Apache Module

"It's easier to find a new audience than to write a new speech." - Dan Kennedy

Now with one of the most intense aspects of the course out of the way, the students are given a task that may have seemed daunting at the beginning of the course. They must reimplement the HTML front-end of MdbLookup server using an Apache module. They compile and install an Apache web server, and write a module to connect to the MdbLookup server. Many students were surprised to find that this is in fact one of the easier labs. It was treated as extra credit.

Software Architecture: The Big Picture

"Details create the big picture." - Sanford I. Weill

This is an appropriate time to reexamine the labs from a software architecture point of view. We show the students how the evolution of MdbLookup reflects the various forms of software architecture:

- Lab 4: command line, confined to the local database
- Lab 5: server, put together using Netcat and pipes
- Lab 6: server, coded using the sockets API
- Lab 7: web-based server, written from scratch
- Lab 8: web-based server, written as an Apache module

The students now have a clear understanding of what a multi-tier client-server architecture entails. They can immediately understand other popular web-based architectures such as Linux-Apache-MySQL-PHP, commonly referred to as LAMP, or Java EE [4]. Our evolutionary approach was constructed conscientiously in order to teach the students how to look at systems from an architectural point of view. This enables the students to dissect a complex software system by identifying the components and understanding the connections between them.

3.5 Resolution

After lab 8, we have about three weeks left in the semester. The students are introduced to C++ at this point. Given the limited time, we focus on one aspect of the language: object construction and destruction. The interplay between memory allocation and object lifetime is often poorly understood, even by those students who use C++ regularly. The lack of a precise understanding of that issue is often the source of bugs. A solid understanding of object lifetime is the foundation for using more advanced C++ features such as inheritance and templates.

Lab 9: Object Construction and Destruction in C++

"High aims form high characters, and great objects bring out great minds." - Tryon Edwards

In lecture, we study a basic C++ string implementation, *MyString*, which has two data members:

```
class MyString {
public:
    ...
    // member functions
    ...
private:
    char *data;
    int len;
};
```

This canonical example of a C++ class shows how constructor, destructor, copy constructor, and assignment operator should perform memory management. We call these member functions the *Basic4*.

Lab 9 helps the students gain a precise understanding of when each of the Basic4 is invoked. We inserted trace output statements in the Basic4 functions of the MyString class. We ask the students to analyze the trace output when the following function is called, identifying precisely at what point each of the Basic4 is invoked and on which object, including unnamed temporaries:

```
MyString add(MyString s1, MyString s2)
{
    MyString temp(" and ");
    return s1 + temp + s2;
}
```

The `-fno-elide-constructors` flag for the `g++` compiler is used in a Makefile that we provide to the students. The flag makes sure that the compiler generates all copy constructor calls; otherwise the compiler will optimize away some copy constructor calls for temporary objects. Lab 9 also asks the students to provide definitions to some of the member functions and operators that the course-provided version left unimplemented.

Lab 10: Linked List Redux

"Experience isn't interesting until it begins to repeat itself." - Elizabeth Bowen

In lecture, we cover the template container classes in the standard C++ library including `vector` and `list`. We emphasize the important semantic difference between these containers and the linked list we implemented in lab 3. The

standard containers provide "value semantics," that is, an item is copied and held by value in the container. In contrast, the linked list from lab 3 only held a pointer to a given object.

The first part of lab 10 tests the students' understanding of this difference. It asks them to write a linked list class in C++, called *StrList*, that holds MyString objects. The challenging requirement is that they need to use the linked list class they implemented in lab 3 as the underlying engine. In other words, the students were required to wrap the lab 3 linked list with a new C++ interface that provides value semantics. For example, one of the member functions that they were asked to implement:

```
void StrList::addFront(const MyString& str)
```

must call:

```
struct Node *addFront(struct List *list, void *data)
```

Being able to switch from value semantics to pointer semantics, and vice versa, represents a significant challenge to the students. As with lab 3, a comprehensive test driver was provided and Valgrind-clean execution was mandatory.

Lab 10 also gives the students a glimpse of working with legacy code. While the first part asked them to wrap a piece of legacy code with a new interface, the second part asks them to do the opposite, namely, keep the new interface and upgrade the implementation. It asks them to turn StrList into *TList*, a template class that can hold data types other than MyString. The underlying engine is no longer based on the lab 3 code, but replaced by the standard C++ `list` template container class. By using the following two typedefs,

```
typedef string MyString;
typedef TList<string> StrList;
```

the students were able to use the same test driver provided in the first part without modification.

SmartPtr: Java-style Object Reference in C++

"We live in an instant-coffee world. Sometimes real-world solutions take a little longer." - Mike Conaway

By this time, the students appreciate the difficulty of object lifetime management in C/C++; they appreciate the benefit of being able to pass around object references in Java without worrying about deallocating the object. We end the course by examining a reference-counted smart pointer class in C++, *SmartPtr*, that can mimic the semantics of Java object references.

A SmartPtr instance is initialized with a pointer to a heap-allocated object–a pointer returned by the `new` operator–and takes over the lifetime management of the given object. The SmartPtr instance is passed around by value. When it is copied, the copy constructor increments the reference count. The SmartPtr's destructor decrements the reference count, and calls `delete` on the underlying object only when the reference count goes to zero. We draw comparisons between a SmartPtr and a Java reference. Although the mechanisms are different (reference counting for a SmartPtr and garbage collection for a Java reference), the effect is the same: you can copy and return an object reference freely without worrying about deallocating the underlying object.

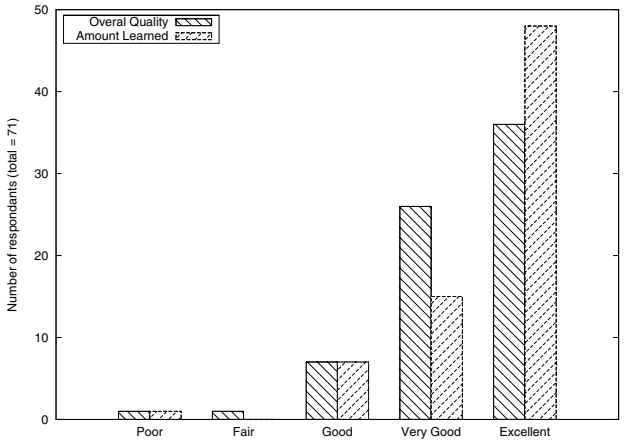

Figure 3: Survey: Overall Quality & Amount Learned

4. STUDENT EXPERIENCE

We taught this course for three consecutive semesters: Spring 2008, Fall 2008, and Spring 2009. The student enrollments were 30, 46, and 49, respectively. The majority of the students were prospective or declared CS majors. Figure 3 shows the results from two relevant questions from the end-of-semester course evaluations. It indicates that student reaction to the course has been overwhelmingly positive.

We implemented several methods to mitigate the demanding workload. We ran a class mailing list, where the students were encouraged to help each other but were forbidden from posting code. This was obviously helpful to those struggling but was equally helpful to those providing guidance as they learned through teaching.

We structured the course rigidly. The lab descriptions clearly laid out the steps needed to complete the assignments correctly. This reflects our viewpoint that, in a C-based introductory systems course, precise direction is more valuable than encouraging creativity. One drawback to this approach was that plagiarism was more difficult to detect.

In order to evaluate our students' performance in the operating systems course (OS), for which our class was designed as preparation, we conducted an additional survey. Among the 40 respondents, 15 went on to take OS. We compared the GPA of those 15 students to that of all OS students from Fall 2008 to Spring 2010, during which all 15 took OS. Our students bettered the four semester average for OS: 3.14 to 2.99. This result is by no means scientific, but we are bolstered by the fact that our students did not perform poorly, especially considering that OS classes are generally comprised of more than 75% graduate students, many of whom have significant industry experience.

Our core mission in education is to engage students and get them excited about CS. This mission, by nature, is difficult if not impossible to evaluate. Perhaps more telling is the personal observations of our students. One student reported that our course "was critical to [his] success in [operating systems]." He went on to describe that many of his peers lacked enough background knowledge in C, and ended up dropping OS. The most gratifying comment we received came from a student who was majoring in Applied Physics, took our course in his third year, and fell in love with CS.

He took as many CS courses as he could in his forth year and was admitted to a top Ph.D. program [15].

> "Due to this course *alone*, I found myself fully-prepared for subsequent coursework in databases, operating systems, and software engineering. [...] We were tacitly taught a lesson in iterative development - we built the server part-by-part, with the requirement of having a working (tested) system at each stage in development. Along the way, I developed skill in efficiently managing a large codebase (in C) on the UNIX command-line - something invaluable in my systems work in graduate school."

5. SUMMARY

We present a one-semester course intended to bridge the gap between a Java-based introductory sequence and advanced systems courses. By anchoring the course on a single real-world application, we were able to provide depth, instill good programming practices, give insight into systems, and generate excitement.

Our contribution is twofold. First, we propose a course organization technique and demonstrate its efficacy. Traditionally classes are structured as either a term project with milestones, or a series of homework assignments testing individual topics. We combine the two approaches. From the outside, our course seems to consist of ten independent assignments, but in fact, they are milestones to a single main project. This technique can be applied to other CS courses. Second, we use this single-project approach to implement a transition course addressing the "gap problem" caused by the shift to objects-first. The course effectively transitioned our students from the introductory Java sequence to upper level systems courses. We present a detailed description of the course material, which others can adapt for their own classes.

Acknowledgments

The authors would like to thank Prof. Angelos Keromytis for providing his previous course materials which gave the foundation for this course, and Prof. Jason Nieh and Prof. Junfeng Yang for their help with the evaluation. We also thank our students and TAs for their effort and feedback.

6. REFERENCES

[1] COMS W3157 Advanced programming, Columbia University. `http://www1.cs.columbia.edu/~jae/cs3157/2009-1/syllabus.html`.

[2] GNU Netcat. `http://netcat.sourceforge.net/`.

[3] GNU Wget. `http://www.gnu.org/software/wget/`.

[4] Java EE at a glance. `http://java.sun.com/javaee/`.

[5] Valgrind. `http://valgrind.org/`.

[6] The Joint Task Force on Computing Curricula. "Computing Curricula 2001". *Journal of Educational Resources in Computing*, 1(3), 2001.

[7] O. Astrachan, K. Bruce, E. Koffman, M. Kölling, and S. Reges. Resolved: objects early has failed. *SIGCSE Bull.*, 37(1):451–452, 2005.

[8] K. B. Bruce. Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list. *SIGCSE Bull.*, 37(2):111–117, 2005.

[9] R. J. Enbody, W. F. Punch, and M. McCullen. Python CS1 as preparation for C++ CS2. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 116–120, New York, NY, USA, 2009. ACM.

[10] J. Forbes and D. D. Garcia. "...But what do the top-rated schools do?": a survey of introductory computer science curricula. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 245–246, New York, NY, USA, 2007. ACM.

[11] J. Gal-Ezer, T. Vilner, and E. Zur. Has the paradigm shift in CS1 a harmful effect on data structures courses: a case study. *SIGCSE Bull.*, 41(1):126–130, 2009.

[12] M. Gancarz. *The UNIX philosophy*. Digital Press, Newton, MA, USA, 1995.

[13] A. Gaspar, N. Boyer, and A. Ejnioui. Role of the C language in current computing curricula part 1: survey analysis. *J. Comput. Small Coll.*, 23(2):120–127, 2007.

[14] R. Hess and P. Paulson. Linux kernel projects for an undergraduate operating systems course. In *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*, pages 485–489, New York, NY, USA, 2010. ACM.

[15] N. Knight. personal correspondence, 2009. UC Berkeley.

[16] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 14–18, New York, NY, USA, 2005. ACM.

[17] R. Lister, A. Berglund, T. Clear, J. Bergin, K. Garvin-Doxas, B. Hanks, L. Hitchner, A. Luxton-Reilly, K. Sanders, C. Schulte, and J. L. Whalley. Research perspectives on the objects-early debate. *SIGCSE Bull.*, 38(4):146–165, 2006.

[18] T. P. Murtagh. Teacing breadth-first depth-first. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 37–40, New York, NY, USA, 2001. ACM.

[19] T. P. Murtagh. Weaving CS into CS1: a doubly depth-first approach. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 336–340, New York, NY, USA, 2007. ACM.

[20] J. Nieh and C. Vaill. Experiences teaching operating systems using virtual platforms and linux. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 520–524, New York, NY, USA, 2005. ACM.

[21] L. Thomas, C. Zander, and A. Eckerdal. Harnessing surprise: tales from students' transformational biographies. In *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*, pages 300–304, New York, NY, USA, 2010. ACM.