Providing Intellectual Focus To CS1/CS2

Timothy J. Long, Bruce W. Weide, Paolo Bucci, David S. Gibson
Computer and Information Science
The Ohio State University
Columbus, OH 43210

Joe Hollingsworth
Computer Science
Indiana University Southeast
New Albany, IN 47150

Murali Sitaraman, Steve Edwards
Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV 26506

{long,weide,bucci,dgibson}@cis.ohio-state.edu
jholly@ius.indiana.edu
{murali,edwards}@cs.wvu.edu

Technical report OSU-CISRC-9/97-TR42
September 1997

Abstract

First-year computer science students need to see clearly that computer science as a discipline has an important intellectual role to play and that it offers deep philosophical questions, much like the other hard sciences and mathematics; that CS is not "just programming". An appropriate intellectual focus for CS1/CS2 can be built on the foundations of systems thinking and mathematical modeling, as these principles are manifested in a component-based software paradigm. We outline some of the main technical features of this approach to CS1/CS2 and report preliminary observations from our experience with it.

1 Introduction

The biology professor can start his first-year course by announcing, "Biology is the study of life. Everything we do in this class is directly related to this theme." What does the computer science professor announce at this point?

Academic computer scientists are of several minds regarding the first-year sequence for computer science (CS) majors, CS1/CS2. The most popular approaches are:

- a "traditional" approach following ACM/IEEE curriculum recommendations, i.e., CS1 comprises an introduction to programming and CS2 an introduction to abstract data types, data structures, and some more advanced algorithms;
- a "breadth-first" approach for CS1 surveying the intellectual landscape of CS, followed by more programming content in CS2; or
- a "software engineering" approach emphasizing software project organization principles and practices.

Perhaps this divergence of opinion reflects the relative youth of CS as a discipline; perhaps diversity here, as elsewhere, is a positive thing; or perhaps none of these is ultimately the right approach to the introductory sequence. CS ought to be competitive with other major academic disciplines in the marketplace of ideas, not just skills — even at the introductory level. Whatever the approach to CS1 and CS2, there should be a fundamental *intellectual theme* which is clearly visible to students, which provides focus for the entire sequence, and which transcends current fashion and thereby stands the test of time.

Whether current versions of CS1/CS2 feature such a clear intellectual basis is a matter of opinion. Here is ours. In the traditional approach to CS1/CS2, textbooks often attempt to weave a unifying theme around one or more of the major programming paradigms. Despite this, it seems that a satisfactory intellectual perspective for the whole of CS does not come across clearly, if at all. In fact, this approach might reinforce the popular view that CS is just about programming. Perhaps those advocating the breadth-first approach hold the same opinion and, thus, their motivation. On the other hand, turning to the breadth of CS for intellectual focus begs an important question. CS students, at some point in their education, must master (among many other things) the content of the traditional approach to CS1/CS2. What will be the intellectual theme for this material when it finally appears? In other words, a survey of CS provides *context* but no immediate intellectual theme for the discipline. A software engineering perspective on CS1/CS2 brings more to the table, such as explicit consideration of the software life cycle and of software process. But does it carry with it an intellectual foundation for CS? We do not see it.

At institutions such as ours it is important not to overlook the practical value of CS1/CS2. Most of our students work at part-time or full-time jobs or internships. They insist that we not stray too far from that which is directly useful. The good news is that CS1/CS2 *can* have rich and foundational intellectual themes even while emphasizing utilitarian principles and practices of software engineering. Pieces of such themes run through many current efforts in CS1/CS2. Our appeal here is to explicitly acknowledge these themes, to integrate and structure the content of CS1/CS2 around them, and to make students explicitly aware of them. In this way, the introductory CS sequence can be intellectually well-founded, for all to see.

In articulating an intellectual theme for CS1/CS2, we concentrate on relating principles and practices of software design to an underlying conceptual (i.e., mathematically well-founded) framework that exposes how we should *think* about software. The central question for the first-year sequence becomes: How should we use this conceptual model of software to help us design software that exhibits essentially desirable properties; i.e., according to what *discipline* should we build software?

At some level all serious work in software, whether in industry or in academia, must address this issue. The remainder of this paper describes how we base the structure and content of CS1/CS2 around a particular conceptual framework and the software discipline it suggests. Aspects of all three of the most popular approaches to CS1/CS2 can be seen in the resulting first-year sequence.

2 Conceptual Framework

What is the central intellectual role of CS as a discipline? We argue that it is *the study and application of languages and methods for making precise and understandable descriptions of things* [11, p. 3]. Traditional programming is one way to describe "how to" do something (an engineer's viewpoint). It is equally important to precisely and understandably describe "what is" (a scientist's and/or mathematician's viewpoint) [1].

2.1 Systems and Mathematical Modeling

Two well-known techniques turn out to be indispensable — indeed, inevitable — in making precise and understandable descriptions of both kinds:

- *systems thinking*, i.e., understanding "things" as "systems" which can be viewed from the outside as indivisible units, or from the inside as compositions of other such systems (a.k.a. subsystems); and
- *mathematical modeling*, i.e., not settling for mere qualitative descriptions of systems, but creating unambiguous formal descriptions using mathematics.

We do not tell students that CS invented these ideas. We point out that the intellectual leverage which CS brings to them is a new degree of *precision* sufficient to communicate complex ideas even to the dumbest machines, and an appreciation for *understandability* that comes from the practical need for ordinary mortals to maintain enormously large and complex software systems.

Example from: Term:	Traditional imperative paradigm	Traditional object-oriented paradigm	Component-based software paradigm
abstract component	procedure or function header	abstract base class, interface, signature, API	abstract template or instance, i.e., specification
concrete component	procedure or function body	class definition	concrete template or instance, i.e., implementation
design-time behavioral relationship	refers to (uses)	refers to (uses), inherits from	extends, implements, encapsulates, ...
composition mechanism	procedure or function invocation	method invocation	template instantiation, operation invocation

Figure 1 — Component-based software terminology, with focus of Section 3 highlighted

2.2 Component-Based Software

Our software design and development paradigm is that of ***component-based software*** [3, 8]. A software component is a description of system behavior, where the "system" might be a complete end-user application or any of its subsystems. An ***abstract component*** describes an outsider's, or ***client***'s, view of a system: a specification. A ***concrete component*** describes an insider's, or ***implementer***'s, view of a system: an implementation. Abstract and concrete components bear various ***design-time behavioral relationships*** [4, 5] to one another as they sit, ready to be used, in a library of reusable software components. Available ***composition mechanisms*** allow clients and implementers to put together components selected from the library, either to create more complex components to populate the library or to build complete applications.

Ordinary programs apparently fit neatly into this terminological framework. Figure 1 illustrates this with mappings for two other popular paradigms as well as for ours. But in traditional approaches, important aspects of software generally are left implicit and imprecise. Specifically, the *behavioral* — as opposed to merely structural — features of abstract components usually are described using wishful naming or, at best, using natural language comments as specifications [10]. The many varieties of design-time behavioral relationships between components usually are not teased out, so the intended meanings of important component relationships remain imprecise or language-dependent. For example, Meyer [7] documents 12 different "forms" (i.e., possible valid uses and therefore meanings) of the "inherits from" relationship that pervades OO software.

Moreover, where precise descriptions do appear, they inevitably take the form of programming language code, which is often hard to understand. At least, it is hard for other humans to understand; "code" works well for communicating with a computer, but unfortunately also for "concealing" information from other humans.

So, we emphasize several objectives in CS1/CS2, all arising from the focus on *precision* and *understandability*:

- viewing everything through the lens of systems thinking;
- using formal mathematical models to make precise the descriptions in abstract components, as well as those in concrete components;
- explicitly and precisely describing design-time behavioral relationships between components;
- reasoning modularly about client usage based on abstract components (i.e., without concern for the concrete components that implement them), and on substitutability properties deducible from the design-time behavioral relationships between components; and
- designing, specifying, and implementing components using a disciplined approach that supports modular reasoning.

These objectives — plus an overriding desire to make all descriptions understandable to other humans — support our view of the intellectual role of CS and set the intellectual theme for CS1/CS2.

3 Behavioral Relationships

Figure 1 highlights one unique technical feature of the component-based software paradigm: its set of very specific design-time behavioral relationships [4, 5]. In this section we discuss three such relationships: *extends*, *implements*, and *encapsulates*. Four others, *uses*, *checks*, *specializes*, and *instantiates*, also are introduced in CS1/CS2.

The RESOLVE discipline [8] is language-neutral but object-based. We have adapted it to encode its principles in C++ [8, 11] and in Ada [5, 6, 8, 9]. Students at OSU and IUS learn the RESOLVE/C++ discipline, while those at WVU learn RESOLVE/Ada. Details that follow are from the RESOLVE/C++ discipline, where most abstract components are encoded as class templates; a few are ordinary classes. Abstract components are like "abstract base classes" in OO, but each abstract component also includes formal model-based behavioral specifications for the provided type and operations [8]. Most concrete components also are class templates; a few are ordinary classes. Concrete components contain implementation details.

We partition CS1/CS2 by the roles students play:

- In CS1, students act as *component clients* who select abstract and concrete components from the library to solve application problems. They compose existing concrete components by template instantiation, thereby gaining access to new types and operations. Then they declare objects of these types and call the operations available for them, writing code that looks like what other C++ programmers might write. The emphasis is on how to use the descriptions in the abstract components (which the concrete components implement) to reason about the behavior of client programs, and on how to extend the functionality of existing components with additional operations.
- In CS2, students continue to select abstract and concrete components from the component library. But now they also act as *component implementers* by encapsulating compositions of existing components into new concrete components. These new components can themselves be put into the library for future use.

There is at least one other important role which students should learn — designing new abstract components — but we do not attempt to practice this in the first-year sequence.

3.1 CS1 Content

Figure 2 shows a typical pattern of design-time behavioral relationships between components as seen in CS1. This figure is called a ***component coupling diagram*** (CCD) because its purpose is to document the coupling between a given component and all other components upon which its behavior depends. Abstract components are shown with rounded corners and concrete components as ordinary rectangles. Design-time behavioral relationships appear as annotated arrows between components.

The *Partial_Map* component family in Figure 2 comprises class templates in a C++ component library which together define what is typically called "symbol table" behavior. One special abstract component (*Partial_Map_Type*) describes the mathematical model of an underlying programming type, and each of several other components (e.g., *Partial_Map_Define*) ***extends*** it by describing an operation to manipulate objects of that type. A ***kernel abstract component*** (e.g., *Partial_Map_Kernel*) bundles together an underlying type with a group of operations which is minimally sufficient to allow a client to observe and control the value of an object of that type. For *Partial_Map* there are five kernel operations.

Each of possibly many concrete components (e.g., *Partial_Map_Kernel_1a*) ***implements*** an abstract component. A client programmer may select any implementation and reason about its behavior in the client program, by referring only to the abstract component which it implements.

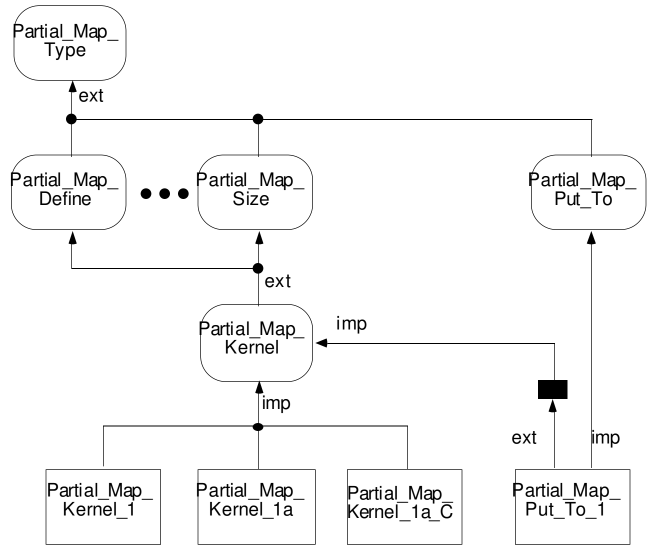

Figure 2 — CS1 design-time behavioral relationships

Often it is useful for a client programmer to create a new reusable concrete component (e.g., *Partial_Map_Put_To_1*) which ***extends*** an existing type with a new operation. Students do this by ***layering*** the code for the extra operation on top of some existing implementation of a kernel abstraction. The small dark rectangle in Figure 2 denotes a formal template parameter to *Partial_Map_Put_To_1* (which is a template). The CCD indicates that the actual parameter supplied by a client when instantiating this template must implement *Partial_Map_Kernel*.

This pattern, or "idiom", illustrates an important use of templates in component-based software: ***decoupling***. The idea is that the code for *Partial_Map_Put_To_1* works with *any* implementation of *Partial_Map_Kernel* — that implementation being chosen by the client, not by the implementer, of *Partial_Map_Put_To_1*. Decoupling every concrete component from *all* other concrete components by using templates in this way maximizes reusability and minimizes code duplication.

By the end of CS1, students have acted as clients of several component families. During the CS1 pilot last year at OSU we used *Natural_Number*, *Set*, *Sequence*, *Queue*, *Stack*, *Record*, *Partial_Map*, and a catalog of simple graphical user interface components.

3.2 CS2 Content

In CS2, a CCD similar to Figure 1 illustrates the other component relationships involved. The student thereby sees the considerable substructure within each ***kernel concrete component*** (e.g., *Partial_Map_Kernel_1*) which implements a kernel abstraction. In particular, such a component ***encapsulates*** some representation data structure (e.g., perhaps a hash table or a binary search tree in the case of *Partial_Map_Kernel*). The bodies of the kernel operations are similar in syntax but different in kind from those encountered in CS1. They manipulate the encapsulated representation by observing a carefully specified ***convention*** (representation invariant) that permits all the kernel operations to cooperate to achieve a certain overall effect. Specifically, what's happening to the representation must lie in proper ***correspondence*** (through the abstraction relation) to the "cover story" in the abstract component which this concrete component implements.

By the end of CS2, students have acted as clients of several additional component families. During the CS2 pilot last year at OSU we added *List*, *Array*, *Binary_Tree*, and several language-processing component families designed for a fairly large project in which the task was to develop a simple logic programming system. Students also have acted as implementers of several component kernels. During the pilot last year at OSU we had them represent a *Queue* as a *List*, a *Sequence* as a *Record* of two *Stacks*, a *Partial_Map* as an *Array* of *Queues* of *Records* (using hashing), *Lists* and *Binary_Trees* using "raw C++" pointers, and several of the logic programming project components as *Binary_Trees* and *Partial_Maps* composed in rather sophisticated ways.

4 Status and Observations

We have been refining this basic approach to CS1/CS2 for a few years at all three of our institutions. We have experimented with many variations, e.g., the degree of formality of specifications, the connections between CCDs and other diagrams and programming language code, the roles of templates, at what point to move students from being clients to being implementers. We have just completed a one-year pilot at OSU and WVU with FIPSE and NSF support, will refine it this year, and expect IUS and a few smaller institutions to pilot our updated course materials during 1998-99. Most of our course materials are available on the web. Please contact the authors for information.

Too few students have completed the pilot sequence so far to provide statistically significant results comparing the new approach with our previous (traditional) CS1/CS2. But based on preliminary examination of formal evaluation data, anecdotal evidence, and instructor feedback regarding student performance and attitudes, we have observed two important positive things about the technical content:

- Students have relatively little trouble learning to use model-based formal specifications to reason about client program behavior [10]. By using pictures and "manipulatives" such as plastic cups and blocks, we are easily able to explain the mathematical theories used in specifications — integers, strings, tuples, sets, trees. Even assertions with quantifiers become understandable with a bit of practice. Perhaps this should not be surprising. After all, these assertions are no more formal than the code which students routinely learn not just to read, but to write, in programming languages. We often provide only formal specifications and *expect* CS2 students to be able to understand them. Most are able to do this without natural language supplements, even under exam conditions.
- Students have relatively little trouble understanding templates. In fact, by the end of CS2 their answers to exam questions and attitudinal surveys indicate not only an understanding of templates but an amazing appreciation of their importance for component-based systems. For example, one student wrote at the end of our CS2 pilot this summer: "What I have learned the most about in this sequence is the use of templates... I never worked with them before this course but now I can't imagine using a language that did not have them." Several students wrote similar comments.

From the experience so far we have concluded that, through a spiral approach, CS1/CS2 students can learn and appreciate the importance of two main driving forces of CS — precision and understandability — and two of the most important techniques which CS offers for achieving them.

We have noticed a few problems with this approach to CS1/CS2, which we currently seek to overcome:

- Some students fail to see why we are not teaching them exactly what (they think) employers want them to know: traditional C++ programming. Especially those few who already have programmed in C++ tend to balk at having to be more rigorous about, e.g., behavioral specifications. So one challenge is to make sure students realize that the *real* CS1/CS2 objective is to lay the foundation for a 35 year computing career, not to providing training for their next co-op job.
- Combining templates and inheritance is relatively new for the popular programming languages we use (C++ and Ada) and, therefore, for their compilers. We have been able to overcome all such problems so far by careful compiler selection and a few workarounds, but the portability of code suffers somewhat as a result.
- It is important to have sophisticated tool support, e.g., for relating CCDs to programming language code. We have been designing improved tools [2] and are now implementing some that we will pilot this year.

We have been able to cover essentially all the traditional CS1/CS2 topics (programming style, algorithms, data structures) that we formerly taught, by using them as the basis for sample components and assignments. By moving gradually from small individual lab projects to two-person groups and larger projects as the year progresses, we motivate the major non-technical as well as technical principles and practices of software engineering. And although we do not consciously seek a breadth-first experience, the effort to provide students with a firm intellectual and philosophical foundation for CS leaves them exposed to a variety of topics not always mentioned in a traditional CS1/CS2: syntax and semantics of natural vs. formal languages, systems thinking, explicit mathematical modeling, HCI issues and GUI components, loop invariants, systematic testing, radix representation of natural numbers, logic programming, etc.

5 Acknowledgment

We gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-9311702, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, or the U.S. Department of Defense.

6 References

[1] Abelson, H., and Sussman, G.J. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, 1985.

[2] Bucci, P. *Conceptual Program Editors: Design and Formal Specification*. Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, 1997.

[3] Edwards, S.H. *A Formal Model of Software Subsystems*. Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, 1995.

[4] [Edwards 97] Edwards, S.H., et al. Software component relationships. In *Proc. 8th Annual Workshop on Software Reuse*, Columbus, OH, 1997.

[5] Gibson, D.S. *Behavioral Relationships Between Software Components*. Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, 1997, to appear.

[6] Hollingsworth, J.E. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada*. Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, 1992.

[7] Meyer, B. The many faces of inheritance: a taxonomy of taxonomy. *IEEE Computer 29*, 5 (1996), 105-108.

[8] Sitaraman, M., and Weide, B.W., eds. Component-based software using RESOLVE. *ACM Software Eng. Notes 19*, 4 (1994), 21-67.

[9] Sitaraman, M. *An Introduction to Software Engineering Using Properly Conceptualized Objects*, WVU Publications, 1997.

[10] Sitaraman, M., et al. *Teaching the Essential Role of Mathematical Modeling in Understanding and Reasoning About Objects*. Tech. report OSU-CISRC-9/97-TR43, Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, Sept. 1997.

[11] Weide, B.W. *Software Component Engineering*, OSU Reprographics, Columbus, 1997.