Components-First Approaches to CS1/CS2: Principles and Practice

Emily Howe, Matthew Thornton, and Bruce W. Weide
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210 USA
+1-614-292-1517
{howe,thorntom,weide}@cis.ohio-state.edu

Technical Report OSU-CISRC-9/03-TR49
September 2003

Components-First Approaches to CS1/CS2: Principles and Practice

Emily Howe, Matthew Thornton, and Bruce W. Weide
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210 USA
+1-614-292-1517
{howe,thorntom,weide}@cis.ohio-state.edu

ABSTRACT
Among the many ways to focus CS1/CS2 content, two have been published that emphasize the concepts of component-based software engineering. Courses based on these two instances of a "components-first" approach are remarkably similar in several crucial respects—which is surprising because they were developed independently and with very different objectives. Indeed, the two versions are based on virtually the same principles for content organization, and they share many common features that are unusual for CS1/CS2. Yet, they are notably different in other ways. Detailed analysis of these similarities and differences suggests that it might be possible to transfer some of their claimed and documented advantages to other approaches within the programming-first paradigm for CS1/CS2, by rearranging the content of such courses in accord with the underlying principles of the components-first approach.

Categories and Subject Descriptors
K.3.2. [**Computer and Information Science Education**]: Computer science education, Curriculum.

General Terms
Algorithms, Design, Languages.

Keywords
Component-based software, components-first, CS1, CS2, objects-first, programming-first, software components.

1. INTRODUCTION
How should we teach CS1/CS2? ACM's *Computing Curricula 2001* describes three substantially different approaches to implementing a programming-first paradigm: imperative-first, functional-first, and objects-first [3]. In this paper, we identify, compare, and contrast two instances of a fourth variant of the programming-first paradigm: **components-first**. This approach has been developed independently by two different groups with different objectives. Despite resulting from completely separate efforts, the two versions largely share the same set of principles that dictate how course content should be organized, and their developers claim similar advantages from this non-traditional organization.

The contributions of this paper are in noting the surprisingly close relationship between the results of these two independent efforts, in highlighting some of the most important similarities and differences with each other and with (especially) the objects-first approach, and in demonstrating thereby that the components-first approach is a distinct approach within the programming-first paradigm. Furthermore, we suggest how CS1/CS2 courses following the other three programming-first approaches might benefit from some content reorganization based on principles of the components-first approach.

2. THE COMPONENTS-FIRST APPROACH
Component-based software engineering (CBSE) is the process of developing a software system, given a significant library of existing software components that are available for "reuse," with the expectation that a substantial part of the resulting system will consist of these components [22]. A closely related notion, *software component engineering*, involves design and implementation of a harmonious component library, such as the Standard Template Library (STL) for C++ [19] and the java.util library for Java [4], to facilitate the practice of component-based software engineering by clients of the components.

The premise of the components-first approach to CS1/CS2 is that the availability of non-trivial software components significantly affects how professionals think about building software. Students might benefit from learning some of these ways of thinking early in the CS curriculum. Indeed, the authors of both versions of components-first note that what made their new courses both viable and necessary was the recent availability of modern component libraries.

Just as CBSE is noticeably different from traditional software engineering, we will see that the components-first approach to CS1/CS2 is noticeably different from the standard programming-first approaches. Historically, component-based software is closely related to object-oriented software. So, among other things, it is natural to ask whether the components-first approach is really so different from the objects-first approach. This will be addressed later, but first it is necessary to introduce the two courses of study that follows the components-first approach.

2.1 Koenig-Moo Version ("K-M")

One version of the components-first approach has been developed by Andrew Koenig and Barbara Moo of AT&T, who wrote *Accelerated C++: Practical Programming by Example* [7] after teaching a short course on C++ at Stanford. In a series of articles in the *Journal of Object-Oriented Programming*, the authors explain the rationale for the organization of their material [8-16]. Their primary objective was not to teach the principles of CBSE, but to teach the C++ language.

The course content organization is based on three main principles that have little to do with C++ itself. Nonetheless, the authors observed that students learned faster and were less frustrated by C++ [7]. These principles are [8]:

- Explain how to use language and library facilities before explaining how they work.
- Motivate each facility with a problem that uses that facility as an integral part of its solution.
- Present the most useful ideas first.

Course content is divided roughly in half. The first half includes how to use the STL and how to apply it to various problems. The second half introduces implementation-level problems: how to define new types (classes), how to use pointers, and inheritance. The book is used at several universities, and is on the recommended reading list at some others.

There is no published systematic evaluation of outcomes when using this material, though in the *JOOP* paper series Koenig and Moo do provide many anecdotes about their experience.

2.2 RSRG Version ("RSRG")

The other version of the components-first approach has been developed by the Reusable Software Research Group (RSRG) at The Ohio State University. The philosophy and some features of this CS1/CS2 sequence have been described over the past several years, primarily at SIGCSE symposia [17,18,1,2]. The objective here was to teach CBSE concepts early in the CS curriculum, with the programming language serving not as the focus but merely as the delivery vehicle. The principles underlying this effort are not stated so succinctly as those above, but from [17,18] it is clear that all three are at work here, too.

Course content is divided roughly in half, as in the K-M version, and along precisely the same lines. However, the technical basis for this version is RSRG's research work on RESOLVE, a framework for writing reusable, component-based software [20]. The RESOLVE research language includes a mathematical syntax for writing formal specifications of components (which is used starting in CS1), as well as constructs for component-based software implementation. There are also language-specific "disciplines" for Ada and C++. The RESOLVE/C++ discipline and component library is used in CS1/CS2 courses at two universities, and various aspects of this version appear primarily in upper-division software engineering courses at some others.

The authors report the results of systematic studies showing significant changes in student attitudes about various software engineering issues [21]. For example, students believed significantly more strongly after CS1/CS2 than before that "it is possible to show that a software component works without actually running it on the computer". Moreover, examination results showed that most CS1 students learned to understand formal specifications up to a few lines long without prerequisite courses in discrete math.

3. THE (SURPRISING) SIMILARITIES

Both K-M and RSRG begin much like other programming-first courses. There is, of course, a good reason for this: students must know how to use certain basic programming language features in order to write even the first interesting program that involves components. Both start with simple example programs involving output, input, `if-then-else` statements, `while` loops, and the built-in scalar types, and eventually use C++ classes and class templates as components.

Other similarities between K-M and RSRG that are more interesting because they are *not* evident in the other programming-first approaches (especially the objects-first approach). Our observations about these features, in the remainder of this section, are based on examination of numerous other CS1/CS2 textbooks and syllabi, as well as on informal discussions with those who teach such courses.

3.1 Client-View-First Pedagogy

The single most significant feature of the components-first approach is that the behavior of a component is understood through its interface, not by looking at source code for an implementation. This leads to many similarities between the two components-first versions in their treatment of traditional CS1/CS2 topics and in content organization, and accounts for many differences with standard programming-first treatments.

A typical objects-first CS1/CS2 course introduces an abstraction such as a stack, perhaps discusses how it can be used in an application, and then discusses how to implement it. Students are alternately clients and implementers of classes—but the client role persists for only a page or two of the textbook and an hour or two of class time.

Students using a components-first approach act as component clients first and component implementers second. In both components-first versions, students are clients for the entire first half of the course and implementers for the second half. Hence, for quite some time students use classes as components to solve application problems, without learning how to implement those or similar classes. This organization is intended to reinforce the role distinction in the minds of students and to make them conscious of the importance of abstraction in explaining component behavior to clients.

3.2 Value Semantics

In all objects-first courses that we are aware of, either the programming language or the inclination of the textbook author or instructor dictates that students rather quickly become aware of the reference-value distinction: some variables hold references to objects, and others hold values. This distinction raises deep technical issues that complicate life for students [10] and software professionals [23] alike.

One advantage of using C++ as a language vehicle (as opposed to, say, Java) is that it is possible to program serious software without making a reference/value distinction. In such an approach, there are only values. K-M achieves this by using the STL, whose developers advise that the default assignment operator and copy constructor should be overridden to make "deep" copies. Indeed, C++ `structs` are introduced early

because `structs` (effectively, records) behave, by default, like values. The RSRG version obtains value semantics in a different way. It prohibits the use of assignment and copy construction for user-defined types (by making these private) and replaces data movement using assignment by data movement using swapping [5,6].

3.3 Pointers

By insisting on clean abstractions for the client view, both components-first versions delay the introduction of pointer-based data structure representations until much later than a traditional CS1/CS2 course. In fact, references and pointers do not arise until students become component implementers in the second half, and then only after layered representations are introduced (i.e., those where the data representation for a new component consists entirely of previously-defined components rather than direct low-level pointer-based structures).

This organization is a direct and natural consequence of thinking in terms of component-based software. Although it is possible to make the conscious decision to introduce layering of classes on top of other classes before exploring low-level data structures, to our knowledge no versions of the objects-first approach actually do so. Similarly, the other programming-first approaches typically do not attempt to keep pointer-based data structures "buried" inside a few low-level components. This is unfortunate for two reasons. First, these concepts are notoriously difficult for many students to comprehend, and even for even professional programmers to "get right" without substantial debugging effort. Second, and more important, software professionals who use modern component libraries such as the STL and java.util rarely need to code their own linked lists or binary trees. These staples are already coded and encapsulated inside a few standard components.

3.4 Loop Invariants

With few exceptions, traditional CS1/CS2 courses do not discuss loop invariants at all, or they treat them as curiosities. We are not aware of any version of the objects-first approach in which loop invariants are used consistently.

By contrast, both components-first versions introduce loop invariants early, while students are still in client-mode, and continue to use them thereafter. The availability of clean abstractions, which are needed for client understanding and reasoning about component behavior, makes this possible [1]. If loop invariants are used only for programs with integers, then students may become convinced that they are suitable only for "toy" programs. But if they are used with input streams and vectors (as in K-M) or with queues, maps, and other components (as in RSRG), then it becomes more plausible that loop invariants are a powerful technique for reasoning about "real" programs, even if students do not practice writing loop invariants themselves.

3.5 Arrays and Other Data Types

Typically, the first moderately complex data type that students encounter in the other programming-first approaches is the array. Part of the reason seems to be that arrays are built-in types in the most commonly used programming languages. (In a functional-first approach, lists usually play the same role, and apparently for the same reason.) If a course seeks to teach a language and arrays are a prominent language feature, then it is quite natural to introduce examples that illustrate it. However, the early introduction of arrays in the objects-first approach is problematic. Because arrays are built-in types rather than class types (in C++ and Java), syntactic sugar distinguishes arrays from all the other complex types that come later. Even with this linguistic support, though, arrays are far from simple. Arrays are parameterized both by other types and by integer sizes or bounds. Moreover, arrays are often presented in terms of their representation as contiguous blocks of memory. This results both in pointers/memory addresses being introduced early and in confounding the natural abstract cover story that an array is like a "table".

In contrast, and in accord with the principles mentioned in Section 2.1, K-M and RSRG do not introduce arrays early, and when they are introduced later, they are introduced for their performance properties rather than their use as a data structure. Their first components—`string` and `Text`, respectively—involve concepts and syntax that apply to subsequent components as well. These types have straightforward abstractions as (mathematical) strings of characters. They serve as simple first examples of "collections" without the need for the complications of type parameters, static bounds on length, knowledge of memory layout, or the introduction of pointers or references. Many interesting application programs need character strings. Both versions of components-first substitute problems/algorithms that involve strings of characters for traditional ones that involve integers and arrays. Students practice early needed programming skills by, for example, writing loops and/or recursive functions that concatenate strings, search for substrings, etc.

3.6 Parameterized Types

After the built-in array, it is common for textbooks and courses taking the objects-first approach to delay or even to completely avoid the introduction of other parameterized types. This is true even when the language (e.g., C++) includes direct support for parameterized types.

In contrast, both K-M and RSRG introduce parameterized types soon after `string` and `Text`, respectively. The K-M version moves directly to the STL `vector`, which is similar to the `Sequence` component that serves as the first parameterized component in the RSRG version. In the latter case, at least two other reasonably complex but not parameterized components are introduced before templates [2].

4. THE (UNSURPRISING) DIFFERENCES

For all their similarities to each other that are unusual compared to traditional programming-first treatments, K-M and RSRG have their differences as well. However, these differences generally can be explained as resulting from the different objectives of the two courses of study. K-M is a C++ course; RSRG is an introductory component-based software-engineering course. These differences help illustrate how to adapt some ideas from components-first to other programming-first approaches to CS1/CS2.

4.1 Language Features

Because of the different objectives of the two components-first versions, they obviously need to cover different language features. For example, K-M starts with `while` loops but eventually introduces all the C++ loop structures, which is consistent with the goal of teaching the C++ language. RSRG sticks with `while` loops alone. The benefits of orthogonality not-

withstanding, this limitation makes it marginally easier to understand, and to be consistent about using, loop invariants.

The two versions differ in their treatments of many other language features, such as exceptions, constants, recommended syntax for template instantiation, etc. Still, K-M introduces language features not willy-nilly but as they are needed to solve problems. Because the component library is the STL, several advanced language features become important along the way. RSRG introduces as few C++-specific language features as possible. Many, however, remain evident despite the design of the component library to avoid complications such as non-default constructors and exceptions. For example, the standard preprocessor idiom that prevents a *.h file from being included multiple times is evident in both versions.

4.2 Level of Formality in Abstractions

As noted earlier, the single most significant feature of the components-first approach is that the behavior of a component is understood through its interface, not by looking at source code for an implementation. This requires that students, during the first half of each version, be able to reason about *what* a component does without knowing *how* it does it. The "cover stories" that explain the values of objects of complex types, and the specifications of the effects of executing methods on them, can be either informal or formal.

K-M takes an informal approach to data type models and specifications. Specifications are not presented in a standard format as they might be if another component library were used (e.g., JavaDoc documentation for java.util). But they are of a comparable nature: English-language descriptions. This is hardly surprising, because formal specifications are not part of C++. RSRG uses formal specifications written in a specification language that augments C++. Several examples of such specifications appear in the literature [1,2,6,17,18]. Because component specifications are important for CBSE in the profession, they are naturally an important topic in the RSRG version of CS1/CS2 because of the overall objective.

4.3 Iterators (or Not)

In dealing with collections, it is necessary to have a mechanism for iterating through a structure. Because it uses the STL, K-M adopts the STL practice of using iterators for this purpose. This implies covering the somewhat unusual syntax and the behavior of STL iterators. Students following RSRG iterate over a collection using an idiom that involves swapping and ordinary `while` loops [5]. This, too, works only because of careful component design, but it involves no new syntax. Despite these differences in detail, the basic concepts students invoke when developing loops are no different in either version of components-first than in the imperative-first and objects-first approaches.

4.4 Inheritance

If Section 3 did not establish that components-first is different from objects-first, the treatment of inheritance seals the case. Both *CC2001* sample implementations of objects-first begin "immediately with the notions of objects and inheritance" [3]. K-M does not mention inheritance until near the end of the book [7], but then treats it much like the objects-first approach. RSRG is quite different. Students use inheritance rather early, but do not understand it as a single language construct. RSRG instead teases out different "acceptable" uses of inheritance and introduces them as needed with their own keywords (by using `#define` in a standard include file), such as `extends`, `implements`, and `encapsulates`.

5. CONCLUSION

We conclude first that *there is* a legitimate fourth programming-first approach to CS1/CS2, namely components-first, and that K-M and RSRG are two instances of it. We characterize it by the use of a library of off-the-shelf components that can be understood without consulting their implementation details, by adherence to the client-view-first ordering of topics, and the observation of the three principles in the K-M approach.

Of course, it is technically possible for anyone to adopt the components-first approach to CS1/CS2 by acquiring the materials for either K-M or RSRG and adapting them to local circumstances. But resistance (of faculty) to such a major change is inevitably great. However, our analysis of similarities and differences for K-M and RSRG suggests several opportunities for CS educators to diversify and perhaps thereby to improve any traditional CS1/CS2 course—without wholesale adoption of the components-first approach.

The most obvious adaptation is to develop a new, small library of components of the sort that are natural for the current approach. These need not be general-purpose like the STL. They can be special-purpose and domain-specific, so long as they are rich enough to be used to solve several application problems that are of interest to the students and are plausibly realistic. Component behavior need not be specified formally unless that is important to the current course/instructor. Given this library, the main change is to reorganize course concepts around these components: to adopt a client-view-first ordering of topics, to motivate client usage by adapting or creating a set of application program examples as lab exercises, and to rearrange the order of introduction of ideas. Some of the benefits attributed to the client-view-first ordering used in components-first might then result.

The suggestion above is, to be sure, still rather radical and effort-intensive in some current CS1/CS2 situations. It involves the creation of a set of components, the client descriptions, and a set of lab exercises, along with the inevitable addition of transitional material. An easier change that might work in other situations is simply to reorganize existing material to respect the client-view-first ordering, without developing any new components. For instance, it is common for a CS2 course to focus on data structures and algorithms. Perhaps it repeatedly introduces a new set of related functions and/or procedures, or a new class, and then immediately delves into how to implement it. The introduction/use and implementation sub-modules within this course can be reshuffled so all or most or all of the uses come first, and only later are the implementation issues addressed. Again, some of the benefits of the client-view-first topic ordering might be expected to result.

There are other possible changes that could leverage the experience of K-M and RSRG but that do not require even this much revision. For example, in a course using C++ or Java, arrays could be deferred until after character strings are introduced and used extensively. The C++ `string` and Java `String` classes are much cleaner than C strings in this regard because their explanations do not involve arrays, memory layout, pointers, null characters, etc. Making this change alone is unlikely to result in significantly different outcomes,

but it might help students more easily make the transition from scalar types to complex types. Or in C++ (alas, not in Java), retaining value semantics as long as possible and deferring pointers until much later might help ease students into the notably difficult issues associated with indirection.

6. ACKNOWLEDGMENTS

We gratefully acknowledge financial support provided to the first author by a diversity grant from the GE Foundation, and that provided to the second author by an REU supplement to National Science Foundation grant CCR-0081596.

7. REFERENCES

[1] Bucci, P., Long, T.J., and Weide, B.W. Do We Really Teach Abstraction? *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education* , ACM Press, 2001, 26-30.

[2] Bucci, P., Heym, W., Long, T.J., and Weide, B.W. Algorithms and Object-Oriented Programming: Bridging the Gap. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2002, 302-306.

[3] *Computing Curricula, Final Draft*. Available at http://www.computer.org/education/cc2001/final.

[4] Deitel, H.M., and Deitel, P.J. *Java: How to Program, Third Edition*. Prentice-Hall, 1999.

[5] Harms, D.E., and Weide, B.W. Copying and Swapping: Influences on the Design of Reusable Software Components. *IEEE Transactions on Software Engineering 17*, 5 (1991), 424-435.

[6] Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience Report: Using RESOLVE/C++ for Commercial Software. *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering* , ACM Press, 2000, 11-19.

[7] Koenig, A., and Moo, B. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, Reading, MA, 2000.

[8] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 1: Goals and Principles. *Journal of Object Oriented Programming 13*, 7 (Nov. 2000), 44-47.

[9] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 2: Two Interesting Decisions. *Journal of Object Oriented Programming 13*, 8 (Dec. 2000), 36-40.

[10] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 3: The First Data Structures. *Journal of Object Oriented Programming 13*, 9 (Jan. 2001), 35-38.

[11] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 4: Emphasizing the Library. *Journal of Object Oriented Programming 13*, 10 (Feb. 2001), 25-27.

[12] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 5: Working With Strings. *Journal of Object Oriented Programming 13*, 11 (Mar. 2001), 29-32.

[13] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 6: Analyzing Strings. *Journal of Object Oriented Programming 13*, 12 (Apr. 2001), 29-32.

[14] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 7: Payback Time. *Journal of Object Oriented Programming 14*, 1 (May 2001), 36-40.

[15] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 8: An Interesting Revision. *Journal of Object Oriented Programming 14*, 2 (June/July 2001), 43-47.

[16] Koenig, A., and Moo, B. Rethinking How to Teach C++, Part 9: What We Learned From Our Students. *Journal of Object Oriented Programming 14*, 3 (Aug./Sep. 2001), 44-47.

[17] Long, T.J., Weide, B.W., Bucci, P., Gibson, D.S., Sitaraman, M., Hollingsworth, J.E., and Edwards, S.H. Providing Intellectual Focus To CS1/CS2. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 1998, 252-256.

[18] Long, T.J., Weide, B.W., Bucci, P., and Sitaraman, M. Client View First: An Exodus from Implementation-Biased Teaching. *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 1999, 136-140.

[19] Musser, D.R., Derge, G.J., and Saini, A. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[20] Sitaraman, M., and Weide, B.W. Component-based Software Using RESOLVE. *ACM SIGSOFT Software Engineering Notes 19*, 4 (1994), 21-67.

[21] Sitaraman, M., Long, T.J., Weide, B.W., Harner, E.J., and Wang, L. A Formal Approach to Component-Based Software Engineering: Education and Evaluation. *ICSE 2001: Proceedings 23rd International Conference on Software Engineering*, IEEE, 2001, 601-609.

[22] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[23] Weide, B.W., and Heym, W.D. Specification and Verification with References. *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, ACM, 2001.