Experience With a Component-Based CS1 and CS2 at Ohio State University

Paolo Bucci, Timothy J. Long, and Bruce W. Weide

Department of Computer & Information Science
The Ohio State University
Columbus, OH 43210

{bucci,long,weide}@cis.ohio-state.edu
http://www.cis.ohio-state.edu/~weide/sce/now

Abstract

Over the past four years, we have been engaged in the design, implementation, and evaluation of a new CS1/CS2 sequence at Ohio State University. In this paper we describe the philosophy that underlies our new courses, how we have attempted to realize this philosophy, and some observations resulting from this experience.

Philosophical Foundations

We start with the premise that software systems should be conceptualized and realized as a composition of components, using off-the-shelf reusable components whenever possible. This is in contrast to a build-from-scratch paradigm. The foundations for this premise arise from the twin pillars underlying all component based engineering:

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

To this starting premise we add the important condition that software systems should support modular reasoning. That is, not only are software systems composed of components, but it must be possible to reason about the behavior of components independently from the systems in which they reside.

Realizing the Philosophical Foundations

Here we describe some of the technical details of software development as we teach it in our classes. These details arise in order to realize the goals of our philosophical foundations.

Software Components

In our courses we use two families of components, abstract components and concrete components. *Abstract* components specify behavior from the outside or from the client perspective. The majority of text appearing within abstract components is normally mathematics. *Concrete* components specify behavior from the inside or from the implementer/maintainer perspective. The majority of text appearing within concrete components is written in a programming language (C++ in our case).

Within the family of abstract components, we use three kinds of abstract components:

- *Abstract type components* describe values that objects can assume. These descriptions are formal descriptions using mathematical modeling. In CS1/CS2 this modeling typically uses such things as mathematical sets, multisets, functions, relations, tuples, strings, trees, and graphs.
- *Abstract extension components* formally describe the behavior of operations using pre- and post-conditions, written in predicate calculus, and using the mathematical models for the types of the operation arguments.
- *Abstract kernel components* are a bundling of an abstract type component and one or more abstract extension components.

Within the family of concrete components, we use four kinds of concrete components:

Each of the components just described are realized using classes and parameterized classes (templates) from the underlying programming language (C++).

Component Relationships

Component relationships are an essential part of the software architecture used in our courses. They explicitly define the relationships between components. We use several component relationships.

- Abstract extension components *extend* abstract type components.
- Abstract kernel components *extend* the abstract type and abstract extension components that they bundle.
- Concrete kernel components *implement* corresponding abstract kernel components and *encapsulate* the data structure representing the corresponding abstract values.
- Concrete extension components *implement* corresponding abstract extension components.
- Concrete specialization components *implement* abstract kernel components and simultaneously *specialize* concrete kernel components.
- Concrete checking components *implement* abstract kernel or extension components and simultaneously *check* concrete kernel or concrete extension components.

Each of these component relationships is realized in the underlying programming language by the inheritance mechanism. However, our explanations, to the students, of these relationships are expressed in terms of the behavioral properties of the components being related and not in terms of how the inheritance mechanism works. For example, if abstract component B *extends* abstract component A, this means that any concrete component that implements B must also implement A.

Decoupling through Component Parameterization

The majority of components in our class catalog are parameterized and thus realized as templates. It seems likely that the most frequent use of templates in CS1/CS2 is to

achieve generalization, especially for container components such as stacks and queues. Perhaps even more importantly, we make heavy use of templates in order to decouple components. By combining this decoupling technique with the separation of components into abstract and concrete families, we are able to achieve an architecture for software components that truly allows for modular reasoning.

As an example, if a concrete extension component B is implementing an abstract extension component A by layering, then the implementation code in B will be calling kernel operations. Rather than explicitly naming the concrete kernel component in B, B is parameterized by a component that implements the kernel. This prevents reasoning for the component B from having to open up and look inside any particular implementation of the kernel. This looking inside would start a chain of reasoning that looks inside other components as well (the subsystems) instead of just looking at the cover stories provided by abstract components.

By way of summary, we have developed a specific architecture for software components whereby software artifacts reflect our philosophical foundations. This architecture, and its philosophical underpinnings, provide the skeleton to which the details of our CS1 and CS2 are attached. An example of this architecture is shown in Figure 1 for a Partial_Map component. Partial_Maps are similar to dictionaries or symbol tables.

Organizational Details

We teach CS1/CS2 as the three-quarter sequence CIS 221, CIS222, and CIS 321. Each quarter is 10 weeks in length. There is a weak programming prerequisite for CIS 221; for example, students need to be able to write simple loops. During autumn, winter, and spring quarters we typically have six sections of CIS 221, four sections of CIS 222, and two or three sections of CIS 321. Enrollments in the summer are considerably less. For all three courses, enrollment per section is capped at 40. The classes are taught by a mixture of regular faculty, lecturers, and grad students.

Component Architecture for Partial_Map

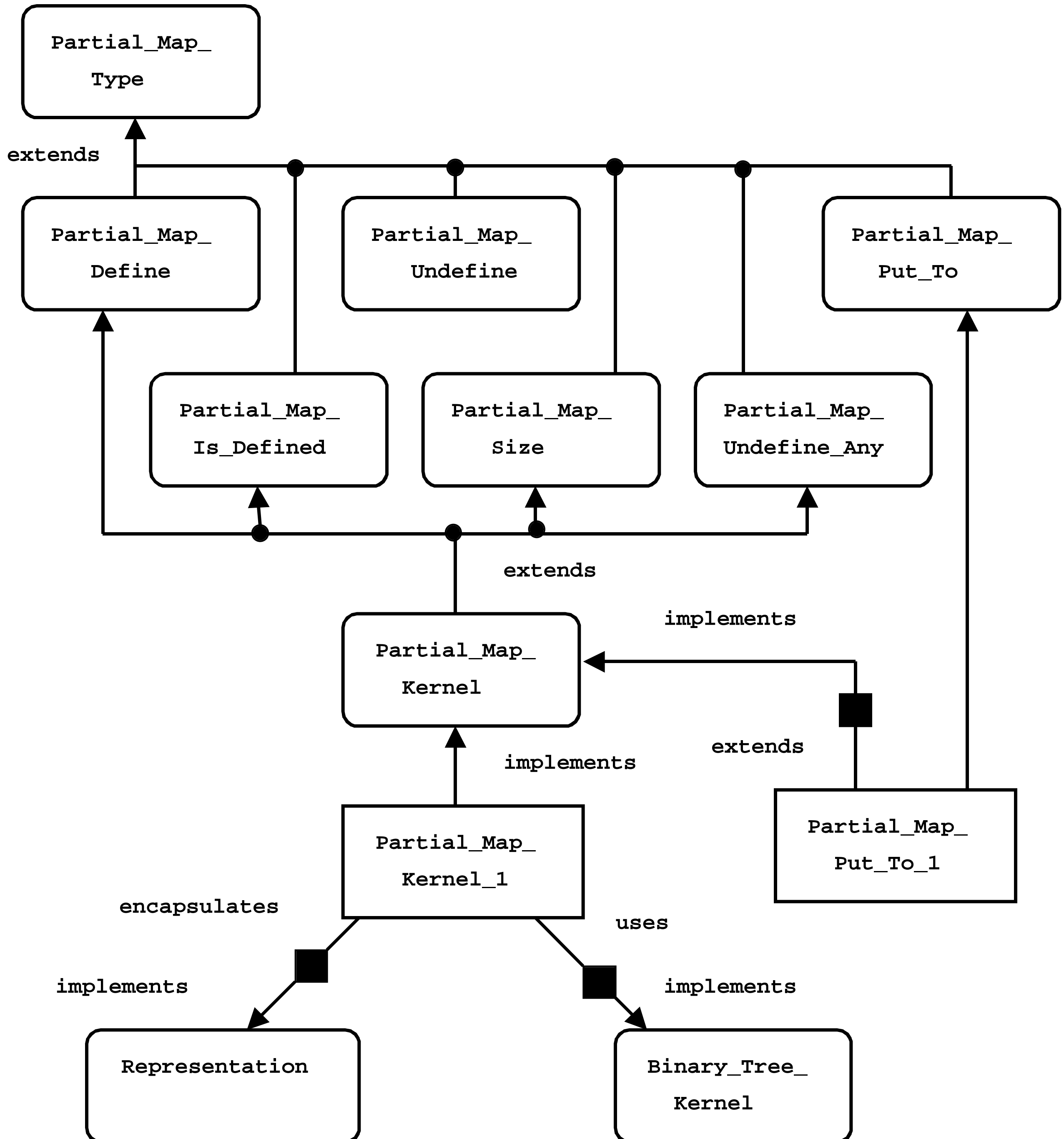

Figure 1

Each course is four credit hours, meeting three times per week for a 48-minute class and meeting once per week for a 48-minute closed lab. In closed lab students do things like implement operations, time algorithms, and participate in debugging and testing contests.

In addition, there are usually six to eight programming assignments per quarter, with each assignment lasting approximately one week.

Course Content

CIS 221

Students act as component clients using components from the class library to solve application problems. All components used are classes and not templates. The emphasis is on how to use the descriptions in the abstract components to reason about the behavior of client programs and on how to extend the functionality of existing components with additional operations.

CIS 222

For the first five weeks of CIS 222, students continue to use components from the client perspective, but now they use parameterized components. They compose existing components by template instantiation, thereby gaining access to new types and operations. They are also responsible for determining which components to use to solve application problems.

During the second five weeks students begin to act as component implementers. They encapsulate data structures representing abstract values and then manipulate the representation to implement operations. At all times students must observe carefully specified representation invariants and carefully specified correspondences between representation values and abstract values.

CIS 321

Two weeks of this course continue with the themes and details from CIS 222. Then the course changes to a project-driven format where students spend eight weeks

implementing a larger project. This involves using existing components and implementing new components as well. The project is designed for students, with supporting material and details of component specifications being discussed in class. The project design conforms in its entirety to our design principles and exhibits the component architecture discussed earlier. Students implement the project in fixed, two-person teams.

Course Pedagogy

CIS 221

This course makes exclusive use of active learning techniques. The material that would normally be delivered in lecture has been written in the form of units. As homework for each class, students read a unit of material and answer questions embedded in the reading as comprehension checks. Upon arriving in class, students work in fixed activity teams of four students each. Together they do a series of pencil-and-paper exercises designed to deepen understanding of the material and to improve their skills. The instructor and grader circulate around the classroom interacting with the activity teams as appropriate. Thus, as students are engaged in those activities that expand their comprehension and improve their skills, they have the advantage of working directly with other students and with the instructor.

It is interesting to contrast this pedagogical approach with traditional lectures. In a lecture format, students listen to a lecture, take notes, and ask questions in class. Then, using notes from class and a textbook, they work individually on homework problems to deepen understanding and improve skills. The key difference, then, between the two approaches are the circumstances under which, and resources available when, students are actively engaged in comprehension and skill building exercises.

CIS 222 and 321

These courses are lecture-based. However, students still work on many in-class activities with their neighbors. Essentially, activities are woven into the lecture format.

In all three courses students work in pairs on all closed lab assignments.

Finally, we make liberal use of manipulatives in all three courses. These include such things as plastic stacking cups that toddlers play with, Leggo blocks, PVC pipes, and even people inside of refrigerator boxes! We use these manipulatives to explain mathematical concepts that students may not be familiar with and to demonstrate the behavior of software components. They are very well received.

Some General Comments

Client View First

This has been an enormous win for us. It is a great aid in helping students separate the two roles of component client and component implementer/maintainer, and the rules of discipline that accompany each role. It makes component-based software possible for students and a reality from the beginning. It even helps with syntax. Even though we use C++, for 10 weeks students deal with very little class syntax because they only see syntax for abstract components. This essentially eliminates the syntactic difficulties associated with teaching objects first.

Exploiting the Intellectual Leverage of Abstraction

To be specific, let us refer to information hiding as keeping internal detail secret from clients. Further, let us refer to abstraction as the process of creating a cover story that explains the hidden details to clients. It seems to us that prior to the new course sequence, the full power of this type of abstraction did not seep deeply into students' experience. However, through the use of mathematical modeling and by initially providing only a client view of components, we believe that our students now experience a radical shift in perspective. Specifically, they now view computation as taking place over mathematical spaces. As an example, Figure 2 shows a tracing table where students will be asked to fill in object values in the right-hand column given execution of the statements in the left-hand column. The table here has been completed as students would

complete it. Notice that the recorded values for the Partial_Map object m are written in standard set notation from mathematics. This is because the mathematical model for Partial_Map is a finite set of ordered pairs obeying the function property; that is, first elements of the ordered pairs are unique. Without coaching, students record the values of m as shown here, even though they have already completed two implementations of partial maps, one using hashing and one using binary search trees.

A Tracing Table

Statement	**Object Values**
	state1 = "Ohio" state2 = "North Carolina" abbrev1 = "OH" abbrev2 = "NC"
object Partial_Map m;	
	m = {} state1 = "Ohio" state2 = "North Carolina" abbrev1 = "OH" abbrev2 = "NC"
m.Define (state1, abbrev1);	
	m = {("Ohio","OH")} state1 = "" state2 = "North Carolina" abbrev1 = "" abbrev2 = "NC"
m.Define (state2, abbrev2);	
	m = {("Ohio","OH"), ("North Carolina","NC")} state1 = "" state2 = "" abbrev1 = "" abbrev2 = ""

Figure 2

The effect shown in Figure 2 is extremely liberating intellectually and can dramatically impact how we conceptualize software. Students' minds are free to consider the rich territory of mathematical spaces for computation instead of being confined to the much more restrictive and fairly sparse world of programming languages.

Students Can Read and Understand Formal Specifications

We have found that students are quite capable of understanding client descriptions of components based on mathematical modeling. Further, there is very little grumbling. As an example, Figure 3 shows a problem from a final exam for CIS 321. We do not ask students to write formal specifications, which is a much harder intellectual task than reading formal specifications. If students learn to write formal specifications at all, that will need to come later in the curriculum when their mathematics skills are stronger.

Problems

We Do Not Ask Students to Design Abstract Components

Based on our experience, students are not ready to design their own components until they have seen several well-designed components, until they have a better appreciation of design issues, and until they have experienced at least one larger, well-designed and well-engineered project. In our sequence, it takes until the end of CS2 for each of these things to happen. Thus, we see the need to have a third course in the sequence where students do design. Of course, it could be that we simply are not clever enough to figure out how to teach design effectively at an earlier stage.

Students Want What is Standard

Students worry that they are not being taught exactly what they think employers want. Some of them balk at being asked to be more rigorous and disciplined about software development. We sometimes have trouble getting them to believe that the real objective of CS1/CS2 is to lay a proper foundation for a 35-year computing career.

Question from a CIS 321 Final Exam

On the next page, implement the following global operation using only BL_Tokenizing_Machine's *kernel* operations.

```
/*!
      math operation TOKENIZATION (
            s: string of character
         ): string of character
         definition
            if s = empty_string
            then
               TOKENIZATION(s) = empty_string
            else
               if there exists a, b: string of character, c: character
                  (s = a * <c> * b and IS_COMPLETE_TOKEN_TEXT(a,c))
               then
                  TOKENIZATION(s) =
                   a * "'" * WHICH_KIND(a) * "\n" * TOKENIZATION(<c>*b)
               else
                  TOKENIZATION(s) = s * "'" * WHICH_KIND(s) * "\n"
   !*/

   global_procedure Tokenize (
         alters   Character_IStream& ins,
         alters   Character_OStream& outs
      );
      /*!
         requires
            ins.is_open = true and
            outs.is_open = true
         ensures
            ins.is_open = true and
            ins.ext_name = #ins.ext_name and
            ins.content = empty_string and
            outs.is_open = true and
            outs.ext_name = #outs.ext_name and
            outs.content = #outs.content * TOKENIZATION(#ins.content)
      !*/
```

Note: The math operations IS_COMPLETE_TOKEN_TEXT and WHICH_KIND were previously defined in BL_Tokenizing_Machine.

Figure 3

Acknowledgement

We gratefully acknowledge financial support from Ohio State University, from the National Science Foundation under grants DUE-9555062 and CDA-9634425, and from

the Fund for the Improvement of Post-Secondary Education under project number P116B60717.

Publications Related to the Course Sequence

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.

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.

Pike, S.M, Weide, B.W., and Hollingsworth, J.E., "Checkmate: Cornering C++ Dynamic Memory Errors With Checked Pointers", *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2000, 352-356.

Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S.M., and Hollingsworth, J.E., "Reasoning About Software-Component Behavior", *Proceedings of the 6th International Conference on Software Reuse*, Springer-Verlag, 2000, to appear.

Bucci, P., Long, T.J., Weide, B.W., and Hollingsworth, J.E., "Toys Are Us: Presenting Mathematical Concepts in CS1/CS2", *Proceedings of Frontiers in Education*, 2000, to appear.

Hollingsworth, J.E., and Blankenship, L., "Experience Report: Using RESOLVE/C++ for Commercial Software", technical report OSU-CISRC-3/00-TR09, Department of Computer and Information Science, The Ohio State University, Columbus, OH, March 2000; submitted for publication.