A Systematic Approach to Teaching Abstraction and Mathematical Modeling

Svetlana V. Drachova, Jason O. Hallstrom Joseph E. Hollingsworth, David P. Jacobs, Joan Krone, and Murali Sitaraman

Technical Report RSRG-11-02
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

September 2011

A Systematic Approach to Teaching Abstraction and Mathematical Modeling

S. Drachova[†], J.O. Hallstrom[†], J. Hollingsworth[‡], D.P. Jacobs[†], J. Krone[♯], M. Sitaraman[†]
[†]School of Computing, Clemson University
[‡]Computer Science, Indiana University Southeast
[♯]Mathematics and Computer Science, Denison Southeast
[†]{sdracho,jasonoh,dpj,murali}@cs.clemson.edu,
[‡]jholly@ius.edu, [♯]krone@denison.edu

ABSTRACT

Abstraction is the process of developing a conceptual veneer that hides the complexity of internals. It is central to computational thinking, in general, and high quality software development, in particular. Use of mathematical modeling makes the abstraction precise. The need for undergraduate CS students to create and understand such abstractions is clear, yet these skills are rarely taught in a systematic manner, if they are taught at all. This paper presents a systematic approach to teaching abstraction using rigorous mathematical models. The paper contains a series of representative examples with varying levels of sophistication to make it possible to teach the ideas in a variety of courses, such as introductory programming, data structures, and software engineering.

This paper presents results from our experimentation with the ideas over a 3-year period at our instiution in a required course that introduces object-based software development, following CS2. The data analysis focuses on students who fall in the bottom half of the performance curve to avoid the bias introduced by top performers, who tend to do well regardless of the teaching approach.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education, Curriculum*

General Terms

Design, Documentation, Reliability

Keywords

Abstraction, mathematical modeling, interface specifications, contract programming, formal methods

1. INTRODUCTION

While long ago, computing graduates may have succeeded by understanding and developing software systems that were entirely in their purview, the next generation of graduates must be equipped to handle the complexity of modern software composed of myriad components, built and maintained by large teams. The development community has seen the conversation shift from systems spanning hundreds of thousands of lines of code, to systems spanning hundreds of millions of lines of code. Indeed, the Software Engineering Institute's Ultra-Large-Scale Systems study projects a future of highly interconnected systems spanning *billions* of lines code [8]. The days of the lone developer with end-to-end system knowledge are gone. Software size and complexity are quickly outstripping the capabilities of a single mind. The next generation of software practitioners must be trained in the principles and skills used to partition and isolate software complexity, and to communicate integration requirements across large teams.

The most fundamental tool for dealing with complexity in any domain is *abstraction*. Neither computational thinking [12] nor software engineering is an exception. Abstraction is the process of developing a simplified conceptual model of a more complex phenomenon. In the context of software development, abstraction results in a conceptual model that explains how a software component is used and what may be expected as a result of its use. This model serves two purposes. First, it hides implementation complexity irrelevant to component clients, including the complexity of other system components. This is why even novice developers typically find it easy to use standard sequencing structures like `queues` and `stacks`; they needn't be concerned with the underlying representations or their interaction with other system components. Second, the model provides a logical partition that gives structure to the development process and enables parallel work to proceed. Put simply, abstraction is the cornerstone of any development paradigm that hopes to tackle large-scale software projects in a cost-effective manner.

Unfortunately, it is not obvious how to teach abstraction. It is largely perceived as an amorphous topic area, rarely taught in a systematic fashion that provides deep coverage of the requisite principles and skills. Even when it is introduced, the rigor of presentation rarely matches the rigor of formal languages students learn in writing programs. In this paper, we focus on reconciling this coverage gap. We describe an approach to teaching software abstraction based

on rigorous mathematical models of software behavior. Students are introduced to a small set of mathematical concepts that are sufficiently expressive to capture the behavior of an interesting set of software components. The state space of each component is documented in terms of these concepts, and the component operations are described by pre-conditions and post-conditions expressed in terms of this state. Within two to three lectures, students are capable of understanding mathematically rigorous specifications — and even develop specifications of their own. We present the approach through representative component examples and present evidence of its effectiveness over a three year period in a sophomore-level object-oriented programming course taught in Java. Our hope is that the paper will serve as a catalyst for broadening the coverage of a skill set that is fundamental to the next generation of practitioners.

Paper Organization. Section 2 briefly surveys related efforts focused on teaching software abstraction and mathematical modeling. Section 3 presents our teaching approach, with an emphasis on representative component examples. Section 4 presents evidence of the approach's effectiveness based on several years of experience with a sophomore-level programming course. Finally, Section 5 concludes with a brief summary of contributions.

2. RELATED WORK

The importance of abstraction in science and engineering has been well understood for many years [7]. The central role that it plays in managing complexity has not been lost on the computing community [5, 12]. Moreover, researchers have noted that students who understand the notion of abstraction are more likely to succeed in computer science [2].

Among the few systematic efforts to teach abstraction is the work of Bucci et al. [3] and Long et al. [6]. The authors emphasize the important role abstraction plays in distinguishing the "client" view of a software component from its implementation. The work of Sooriamurthi [11] aims to teach the basic principles of abstraction to beginning programmers. Unlike these prior efforts, the approach presented here is adaptable and incremental. It may be used in introductory or later courses. The emphasis here is on teaching abstraction through a series of mathematical models, as well as in assessing specifically how well students understand abstraction using these models.

3. APPROACH

Our instructional approach is guided by the supposition that abstraction and modeling are fundamental skills, which, if taught early, benefit students throughout the curriculum and beyond. Accordingly, we make few assumptions about the target audience. We assume prior exposure to three areas: (*i*) basic discrete structures, including sets and tuples; (*ii*) boolean logic, including the primitive connectives; and (*iii*) basic imperative programming. It is important to emphasize that we assume only *exposure* to these topic areas, not in-depth knowledge. As a result, the instructional approach is directly applicable to most CS2- and CS3-level courses, and with some additional background coverage, may even be suitable for a CS1-level course.

When introducing students to abstraction and modeling, we begin by highlighting the observation that if they have ever written a successful program, they must have already practiced these skills at some level. Consider, for example, the following simple code fragment.

```
int i = ... ;
int j = add(i, 5);
System.out.println(j);
```

When asked to explain what this code fragment does, most students immediately respond that it displays the value of `i` plus `5`. When asked to explain how this conclusion was reached, students appeal to integer arithmetic, revealing their implicit use of abstraction and modeling for the `int` data type with mathematical integers. After all, we remind our students, a computer doesn't really store mathematical integers; it stores some representation of the implicit model they have in their minds. Yet even for simple types like `int`, implicit models can lead to reasoning difficulties. The immediate explanation for the short code fragment is incorrect; few students account for `MAX_INT`, the largest possible `int`. The example is trivial, but illustrates a powerful lesson for students: Informal, implicit models of software behavior are often insufficient; they tend to break down at the boundaries and introduce reasoning errors.

The difficulties students encounter when considering the above code fragment can be remedied by making their implicit model of the software explicit. The particular syntax used to achieve this is not critical; a variety of notations may be used. Here we adopt a variation of the RESOLVE specification notation [10], adapted for use in Java:

```
// type int is modeled by: integer
// constraint:  MIN_INT <= self <= MAX_INT
// initial value: 0
```

The specification fragment documents the model for `int`. It specifies that all `int` variables may be thought of as mathematical integers. The associated `constraint` makes the integer bounds explicit. The `self` keyword refers to an arbitrary `int` value; all such values must be between `MIN_INT` and `MAX_INT`, platform-dependent constants. Finally, the `initial value` clause indicates that every `int` will be initialized to zero at the point of declaration.

Models of this type support a modest improvement in students' ability to reason about software by making implicit assumptions explicit. They are, however, insufficient by themselves to enable rigorous reasoning. Students must also have a precise understanding of how the operations within their software influence these models. There are two aspects to consider. First, students must understand their obligations as callers; when can each function be invoked? These requirements are referred to as "pre-conditions"; they are state requirements, expressed in terms of the relevant mathematical models, which must be satisfied before an operation is called. Second, students must understand the guarantees provided by each function; what will they do if called correctly? These conditions are referred to as "post-conditions"; they are state assertions that an operation guarantees will hold upon termination, assuming its pre-conditions were met. This style of programming, based on pre-conditions and post-conditions, is generally referred to as "contract programming" (or "design-by-contract"). Again, we remind our students, even if they're hearing this terminology for the first time, the ideas should be familiar. We are only making explicit a process students must be using at some level if they

are able to write correct software. Consider the specification of `add()` from our example:

```
public static int add(int i, int j) { ... }
// requires: MIN_INT <= (i + j) <= MAX_INT
// ensures: (i = #i) and (j = #j) and
//          add() = (i + j)
```

The `requires` and `ensures` clauses specify the function's pre-condition and post-condition, respectively. Each is represented by a mathematical assertion defined in terms of the `int` model. The pre-condition must evaluate to *true* before the function is called, and the post-condition must evaluate to *true* after the function returns. The pre-condition states that the sum of the integers `i` and `j` must be within the bounds of the model (else, presumably, there will be an overflow within the implementation of the operation.) The post-condition is a conjunctive assertion consisting of three clauses. The first two state that the arguments to `add()` will be unchanged[1]. The `#` notation is used in post-conditions to refer to the pre-conditional values of variables. The final clause states that the return value of `add()` will be equal to the sum of `i` and `j`.

Equipped with the formal model of `int` and the specification of `add()`, students are able to reason precisely about the code fragment. In doing so, they quickly identify a problem: If `i` is assigned `MAX_INT`, the pre-condition of `add()` will be violated. As a result, there are no guarantees! An interesting exercise asks students how the code fragment might be modified to safeguard against an illegal call to `add()` when `i` is sufficiently large. An interesting follow-up exercise asks how the specification of `add()` might be modified to allow arbitrary argument values. These exercises reinforce the relationship between specifications and implementations and demonstrate that software correctness is about ensuring a match between them.

Before considering more interesting component examples, it is useful to observe that the type of preservation constraint expressed in the post-condition of `add()` is common. We often need to state that a particular argument, say `arg`, is preserved by some function. To avoid cluttering the specifications with clauses of the form `(arg = #arg)`, we introduce a short-hand notation. Specifically, we introduce an optional `preserves` clause that captures a comma-separated list of variables preserved by the corresponding function. We remind students that this isn't a new type of assertion. It is simply a short-hand for expressing a common condition.

3.1 Real World Modeling Examples

To further motivate mathematical modeling and abstraction, we have also found it useful to consider "real world" examples familiar to students. In one exercise, students are asked to develop the formal model of the state of a traffic light. A number of possible solutions typically arise, including a triple of booleans, a pair of booleans, and an integer. Each of these models requires a constraint to limit the state space to three possibilities (i.e., red, green, yellow), similar to the constraint placed on the integer model for the `int` type. Whereas the boolean triple requires a constraint that only one of the booleans be true, the integer version must restrict the value to be within 1 and 3. Once a model is identified, the specifications of operations such as `turnRed()` or `isRed()` can be developed. This exercise serves as a catalyst for a discussion of what it means for one model to be "better" than another. It also raises the point that model selection depends on the desired set of operations.

Another useful discussion considers the modeling space. For example, a suitable model for the state of a paper weight might simply be a real number; but depending on the problem, the dimensions may also need to be captured.

3.2 Component Examples

Whereas the examples presented thus far are suitable for any introductory course, the examples in this section focus on common data structures found in standard class libraries (e.g., stacks, queues, lists, sets)[2]. Consider, for example, documenting the interface of a stack component that provides a minimal set of operations (i.e., `push()`, `pop()`, `length()`, `clear()`). To motivate the need for a mathematical model, it is useful to ask students to work in pairs and to pretend that one member of the pair has never heard of a stack. The other member is tasked with carefully explaining how the data structure works without assuming a particular implementation and without using any self-references (e.g., "A `stack` is like a stack of ...", "`push()` pushes an element..."). This is surprisingly difficult. Students' descriptions are likely to be both verbose and ambiguous. The exercise demonstrates that natural languages are a poor choice for documenting software operations, which have precise usage requirements and associated behaviors.

To specify the `Stack` interface, a mathematical model must be selected that captures the state space in a way that is descriptive to developers and supports the documentation of pre-conditions and post-conditions. A mathematical *string* is a good choice. (A useful discussion asks why a *set* would be a bad choice.) A string over some set of elements —the same notion as Σ^* over an alphabet Σ used in formal languages courses— is an ordered sequence of elements expressed as a comma-separated list enclosed within angle brackets. The string $< 4, 6 >$, for example, contains two integers; $<>$ denotes the empty string. The model is a convenient first example because, in addition to the notion of an empty string, there are only two string operations, *length* and *concatenation*. The length operator is unary, represented as vertical bars placed on either side of the argument. The expression $| < 4, 6 > |$, for example, evaluates to 2. Concatenation, the second operator, is *binary*, represented by an asterisk between its arguments. The expression $< 4, 6 > * < 1 >$, for example, yields the string, $< 4, 6, 1 >$[3].

With this background in place, students are prepared to consider the model of the `Stack` interface:

```
public interface Stack<E> {
// Stack<E> is modeled by: string of E
// initial value: <>
```

The `modeled by` clause states that each `stack` should be thought of as a mathematical string of `E`, where `E` is a

[1]Note that in Java, it is impossible for `add()` to modify its arguments since they must be passed by value. Still, we include these preservation conditions for the sake of completeness since they are useful in the context of other languages.

[2]These models are not meant to be novel; variants are routinely found in the specification literature.

[3]It is important to make clear that these mathematical strings are quite different from the `String` type in Java.

generic type parameter; it could be anything[4]. The `initial value` clause states that all `stacks` are initially empty.

Now consider the interface operations, beginning with the specification of `push()`:

```
void push(E e);
// requires: true
// preserves: e
// ensures: self = <e> * #self
```

The pre-condition is trivial; it indicates that `push()` does not impose any calling requirements. (In later examples, we will omit such pre-conditions by eliding the `requires` clause.) The `preserves` clause states that the argument `e` will not be modified by the operation. Finally, the post-condition states that upon return, the final value of the `stack` (i.e., its *post-conditional value*) will be equal to the string containing the argument `e` concatenated with the value of the `stack` prior to the call (i.e., its *pre-conditional value*). It makes no difference where the new entry is specified to be placed, as long as the specification of `pop()` is consistent. Note that the `self` keyword is analogous to `this` in popular languages like Java and C^{++}. It refers to the "current" instance. The key difference is that `self` refers to the instance's model.

Next consider the specification of `pop()`:

```
E pop();
// requires: |self| > 0
// ensures: #self = <pop()> * self
```

The pre-condition states that `pop()` may not be called on an empty `stack`. (A useful exercise asks students to develop an alternative expression of this requirement — e.g., `self != <>`.) The post-condition states that the string containing the element returned by the function, concatenated with the post-conditional value of the `stack` will be equal to the `stack`'s pre-conditional value.

Next consider the specification of `length()`:

```
int length();
// preserves: self
// ensures: length() = |self|
```

The specification states that `length()` will not modify the target object, and it will return the length of the `stack`.

Finally, consider the specification of `clear()`:

```
void clear();
// ensures: self = <>
```

The specification states that `clear()` resets the `stack` to its initial value, the empty string.

One surprising outcome of introducing this material in introductory courses is that most students are able to read and understand specifications of similar complexity almost immediately. Even more surprising, many are able to write specifications after only one or two lectures. A good first exercise asks students to work in pairs to develop the formal specification of a queue component that provides a minimal set of operations (i.e., `enqueue()`, `dequeue()`, `length()`, `clear()`). The most obvious solution mirrors the `Stack` example, but requires `dequeue()` to operate on the opposite side of the string as `enqueue()`. An equally appropriate solution concatenates the new entry to the end of the string. Beyond reinforcing basic modeling skills, the exercise underscores that the specification is merely a "cover story". What matters is that the specifications be meaningful when the operations are considered in their totality.

To introduce students to the modeling techniques used to describe composite types, we next consider the `SplitList` interface. `SplitList` represents a partitioned list of elements. The interface provides operations for moving the list partition and accessing the element to its right. Formally, `SplitList` is modeled as a pair of strings:

```
public interface SplitList<E> {
// SplitList<E> is modeled by:
//   (l: string of E, r: string of E)
// initial value: (<>,<>)
```

The model designates the first element of the pair as `l`, for "left", and the second element of the pair as `r`, for "right". Initially, both strings are empty. The interface provides three operations for moving the list partition:

```
void moveToFront();
// ensures: self.l = <> and
//          self.r = #self.l * #self.r

void moveToEnd();
// ensures: self.r = <> and
//          self.l = #self.l * #self.r

void advance();
// requires: |self.r| > 0
// ensures: (self.l * self.r =
//           #self.l * #self.r) and
//          |self.l| = |#self.l| + 1
```

Consider a list with the value $(< 1, 3 >, < 9, 1 >)$. The first operation, `moveToFront()`, moves the partition to the left side of the list. According to the post-condition, this empties the left string and prepends its elements to the right string — $(<>, < 1, 3, 9, 1 >)$. The second operation, `moveToEnd()`, is similarly defined, but moves the partition to the right side of the list — $(< 1, 3, 9, 1 >, <>)$. The final operation, `advance()`, is more complex. The pre-condition requires that the operation be called only when the right string is non-empty. The post-condition states that the contents of the left and right strings taken together remain unchanged by the operation, but the length of the left string is increased by one. It essentially states that an entry is moved from the right to the left string. An alternative expression of this post-condition may be used to introduce existential quantifiers, depending on the maturity of the students.

The next two operations support addition and removal by providing access to the left side of the right string:

```
void addElement(E e);
// preserves: self.l, e
// ensures: self.r = <e> * #self.r

E removeElement();
// requires: |self.r| > 0
```

[4]The generic parameter `E` will eventually be replaced with an actual programming type, say `Integer`. This type will also be specified mathematically. When its mathematical type is substituted for `E` in the `modeled by` clause, a *fully instantiated* model will result, namely, a string of integers.

```
// preserves: self.l
// ensures:
//   #self.r = <removeElement()> * self.r
```

The `addElement()` and `removeElement()` specifications are immediately understandable to most students since they mirror the `push()` and `pop()` specifications they have already seen. The remaining methods— `leftLength()`, `rightLength()`, and `clear()`— are omitted; their specifications are straightforward.

With the limited exposure to formal abstraction and modeling that these examples and suggested exercises provide, students are able to read and understand a surprisingly broad set of component specifications. One of the more complex examples involves the `Sequence` interface, used to model an unbounded, indexed sequence of elements, similar to Java's `List` interface. The approach to specifying indexing behavior is particularly important given its applicability to several other common data structures. Consider the partial `Sequence` specification:

```
public interface Sequence<E> {
// Sequence<E> is modeled by: string of E
// initial value: <>

void addElement(int pos, E e);
// requires: 0 <= pos <= |self|
// preserves: pos, e
// ensures:
//   there exists l,r : string of E
//   s.t. #self = l * r and
//        |l| = pos and
//        self = l * <e> * r

E removeElement(int pos);
// requires: 0 <= pos < |self|
// preserves: pos
// ensures:
//   there exists l,r : string of E, e : E
//   s.t. #self = l * <e> * r and
//        |l| = pos and
//        self = l * r and
//        removeElement() = e

  ...  remaining operations elided ...
```

The interface is modeled as a string, and instances are initially empty. The pre-condition on `addElement()` indicates that an element may only be added adjacent to an existing element, or at position zero when the string is empty. The first two clauses of the post-condition state that the pre-conditional string can be partitioned into two substrings, `l` and `r`, where the length of the first substring is equal to the desired insertion point. The final clause states that the post-conditional string will consist of the same two substrings, with the argument `e` inserted between them. The specification of `removeElement()` is similar. An interesting question to raise with students asks why the post-condition of `removeElement()` could never be satisfied if the pre-condition were weakened to `0 <= pos <= |self|`.

The presentation has focused primarily on string-based abstractions. It is important to emphasize, however, that the same approach can be used across a wide range of mathematical models and software components. Consider, for example, the partial specification of a basic `Set` interface:

```
public interface Set<E> {
// Set<E> is modeled by: finite set of E
// initial value: {}

void add(E e);
// requires: e ∉ self
// preserves: e
// ensures: self = #self ∪ {e}

void remove(E e);
// requires: e ∈ self
// preserves: e
// ensures: self = #self - {e}

public boolean isIn(E e);
// preserves: self, e
// ensures: isIn() = (e ∈ self)

  ...  remaining operations elided ...
```

The interface is modeled as a finite set with an initial value of empty set. The `add()` operation is used to add an element, assuming it was not a member of the set prior to the call; `remove()` is analogously defined. The `isIn()` operation supports membership testing. An interesting exercise asks students to consider the omission of the pre-condition on `add()` and the associated impact on the performance of possible implementations of `add()` and `remove()`.

We note that these specifications are unremarkable — which is itself remarkable! The point is that by selecting mathematical models that are well matched to the interface abstractions, the resulting specifications are generally straightforward, even when the components are themselves rich. In our own classes, we introduce students to models involving strings, sets, tuples, functions, and other structures, with nearly uniform positive results.

3.3 Example Exercises

We conclude the overview of our approach with a brief summary of three kinds of exercises used to reinforce these abstraction and modeling skills. The simplest asks students to develop test cases for a `mystery()` operation based only on an associated model and mathematical pre- and post-conditions of an operation. A more challenging question asks students to write a formal specification given a formal model, an *informal* natural language description of the `mystery()` operation, and a series of valid test cases (i.e., input and output values). An even more challenging question asks students to devise a mathematical model for a new software component. Of course, these are only samples.

4. EVIDENCE FOR LOWER-PERFORMERS

Over a period of three years, in a sophomore-level, object-based Java course, we have collected evidence that undergraduate students are able to understand mathematical abstraction, as exhibited by their ability to read, write, and use specifications based on these abstractions. To avoid the bias introduced by top performers, who tend to do well regardless of the instructional approach, we focus our analysis on the bottom half of the performance curve, based on graded assignments and exams. At the end of each semester,

these students were given summative assessments that test specific skills required to think abstractly, model data structures with a variety of models, and understand and develop mathematically rigorous specifications.

The most basic questions ask students to create test points (i.e., pre-conditional and post-conditional values) based on their understanding of a component's mathematical model. The most complex ask them to apply their knowledge by developing formal proofs of code correctness that involve application of theorems from relevant mathematical theories. Other interesting questions ask them to develop formal specifications based on these models:

```
public void mystery(Sequence s1,
       Sequence sub, Sequence s2, int p);
```

requires:

- `sub` must be a substring of `s1`
- `p` must be less than or equal to the length of `s2`

ensures:

- `sub` will be removed from `s1`; no other changes will be made to `s1`
- `sub` will not be modified; it will be inserted at position `p` within `s2`
- no other changes will be made to `s2`
- `p` will not be modified

example:

#s1=<1,2,3,4,5>; #sub=<3,4>; #s2=<1,2,3>; #p=1
s1=<1,2,5>; sub=<3,4>; s2=<1,3,4,2,3>; p = 1

Q2. Provide the formal pre-condition for this method.
Q3. Provide the formal post-condition for this method.

The assessment questions are based on the *Reasoning Concept Inventory* (RCI) that has evolved over several years to capture the analytical reasoning skills computing graduates must possess to develop and maintain high quality software [4, 9]. The results of our assessment are summarized in the table below. Subsequent columns represent the average performance on the concept, the number of students scoring above 70%, and the corresponding percentage, respectively. The results span 4 sections of an introductory programming class offered over the course of 3 years. A total of 94 students completed the class; the top-scoring 44 were exempted. The analysis pertains to the remaining 50.

Concept Item	**Avg.**	#≥ 70%	%≥ 70%
Specification basics	80%	30	81.1%
Reading specifications	97.7%	48	96%
Writing pre-conditions	70.8%	28	56%
Writing post-conditions	56.6%	23	46%
Reasoning basics	80%	29	78.4%
Proof assumptions	90%	33	89%
Proof obligations	73.6%	24	64.9%
Correctness proofs	22.5%	8	21.6%

Rows 1–4 correspond to reading and writing mathematical specifications. Rows 5–8 correspond to applying specifications to support reasoning. Naturally, students found writing specifications (as in the exercise above) to be more difficult. Students had the most trouble with proofs. This data informs us that while we have been successful in teaching mathematical abstraction and its use in reasoning to lower-performing students, we still have work to do in teaching them how to write specifications and proofs, either in this course or elsewhere (e.g., discrete structures).

5. CONCLUSION

The point of departure for our work is best summarized by a well-known quote from Aho and Ullman [1]: *"Computer Science is a science of abstraction — creating the right model for a problem and devising the appropriate mechanizable techniques to solve it."* The quote reflects our belief that abstraction and modeling are central to computing; every CS graduate should be able to demonstrate mastery in these skills. Yet surprisingly, little emphasis has been placed on teaching these skills in a systematic fashion.

We have described a systematic approach to teaching abstraction based on formal interface models. We have presented representative interface examples and classroom exercises and summarized our experiences teaching these skills in our courses. The ideas can be integrated in a variety of CS courses. We have shown that lower-performing students can be taught abstraction, and we are in the process of replicating our efforts at other institutions.

Acknowledgments

This work is supported by the National Science Foundation through awards DUE-1022941 and CNS-0745846.

6. REFERENCES

[1] A. Aho and J. Ullman. *Foundations of Computer Science*. W. H. Freeman & Co., New York, NY, 1994.

[2] J. Bennedssen and M. Caspersen. Abstraction ability as an indicator of success for learning computing science? In *The 4th ICER*, pages 15–26, New York, NY USA, September 2008. ACM.

[3] P. Bucci, T. Long, and B. Weide. Do we really teach abstraction? In *The 32nd SIGCSE*, pages 26–30, New York, NY USA, February 2001. ACM.

[4] J. Krone et al. A reasoning concept inventory for computer science. Technical Report RSRG-10-01, Clemson University, September 2010.

[5] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, MA USA, 1st edition, 2000.

[6] T. Long, B. Weide, P. Bucci, and M. Sitaraman. Client view first: An exodus from implementation-biased teaching. In *The 13th SIGCSE*, pages 136–140, New York, NY USA, March 1999. ACM.

[7] D. Norman. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. Addison-Wesley, Boston, MA USA, 1993.

[8] L. Northrop et al. Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon, June 2006. www.sei.cmu.edu/uls/downloads.html.

[9] RSRG, Clemson University. Reasoning concept inventory, 2011. resolve.cs.clemson.edu/wiki/.

[10] M. Sitaraman and B. Weide. Component-based software using Resolve. *SIGSOFT SEN*, October 1994.

[11] R. Sooriamurthi. Introducing abstraction and decomposition to novice programmers. In *The 14th Annual ITiCSE*, pages 196–200, New York, NY USA, July 2009. ACM.

[12] J. Wing. Computational thinking. *Communications of the ACM*, 49:33–35, March 2006.