Abstraction Abstracted

Russ Abbott
Department of Computer Science
California State University
Los Angeles, Ca. USA
01-323-343-6690
Russ.Abbott@GMail.com

Chengyu Sun
Department of Computer Science
California State University
Los Angeles, Ca. USA
01-323-343-6690
csun@calstatela.edu

ABSTRACT
An abstraction is the reification of and conceptualization of a distinction. We use the process of forming abstractions to make sense of the world, i.e., to form concepts. Once created we are often able to externalize these concepts as software. Abstractions are what give software elegance. Abstractions build on each other, producing a hierarchical dependency structure that often creates challenges for understanding. We can teach the use of pre-packaged abstractions. It is more difficult to teach the self-awareness necessary for inventing new abstractions. The process of building abstractions is bottom-up. Thought externalization is where top-down meets bottom-up.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Software as the externalization and embodiment of abstractions.

General Terms
Design, Languages.

Keywords
Abstraction, conceptual model, mental model, refactoring, self-awareness, thought externalization.

1. INTRODUCTION
Liskov [11] describes what she claims are the two kinds of abstraction: abstraction by parameterization and abstraction by specification. Abstraction by parameterization covers the traditional forms of software abstraction, including procedural abstraction (procedures, functions, etc.), data abstraction (data types, classes, etc.), control abstraction, and polymorphic abstraction.

Abstraction by parameterization extracts an essential core of some computational element and reifies it as a named element of its own, leaving parameters to be filled in when the abstraction is instantiated. The while-loop construct, for example, abstracts the notion of iteration and by providing slots into which a Boolean expression and a code segment may be inserted makes iteration available generically. When used wisely, abstraction by parameterization can endow software with elegance, grace, and beauty. Section 2 discusses software elegance. Sections 3 - 6 discuss the application of abstraction by parameterization to software.

According to Liskov, "Abstraction by specification abstracts from the implementation details (how the module is implemented) to the behavior users can depend on (what the module does)." The implication is that one finds a piece of code, figures out what it does, and then expresses that understanding as an abstraction. Occasionally this may actually occur, but in most cases, the process goes in the opposite direction. One knows the behavior one wants and then writes code that implements it.[1] How does one know what one wants? The flippant answer is that one makes it up. But in fact, that is what happens. One determines what one wants by building mental models of the world, real or imagined.

Encarta[2] lists the following as one of the senses of *abstraction*.

> PHILOSOPHY **conceptualization**: the philosophical process by which people develop concepts either from experience or from other concepts.

Although labeled the "philosophical sense," conceptualization captures the general meaning of abstraction. When an entity, biological or mechanical, makes a distinction (e.g. by behaving differently under different circumstances) it has not created an abstraction. An abstraction is a reified distinction, a distinction for which the entity has a name and to which it can refer—abstractly. Abstraction implies conceptualization, which implies a mind that holds the abstraction.[3] The ability to do abstraction as conceptualization is characteristically human.

Abstraction as conceptualization is relevant to software engineering when we use it to understand an application domain and to write requirements as a first step in developing a software system. Section 7 discusses this issue.

So I agree with Liskov that abstraction—somewhat reformulated—plays two important roles in software engineering. Abstraction allows developers to write better software. And abstraction allows analysts to build mental models of the world, which can then be externalized, first as requirements and then as the software that expresses and animates the abstractions.

2. SOFTWARE ELEGANCE
I would guess that most of the participants in this workshop think of themselves—undoubtedly with justification—as among those whom Kramer and Hazzan [10] single out as being

---

[1] Of course, this is grossly oversimplified. Most software development is much more iterative.

[2] http://encarta.msn.com/dictionary_/abstraction.html.

[3] I am not proposing a theory of mind. I am taking our common experience of having minds as primitive.

> able to produce clear, elegant designs and programs … *Abstraction* is suggested as the source for this phenomenon.

Some software is simply far more graceful and conceptually elegant than other software. Software is art—or at least many of those most accomplished in software development think so. In Knuth's 1974 Turing Award lecture [9] "Computer Programming as an Art" he described his feelings.

> [W]hen we prepare a program, it can be like composing poetry or music …
>
> The possibility of writing beautiful programs … is what got me hooked on programming in the first place. …
>
> Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write *grand* programs, *noble* programs, truly *magnificent* ones!

As a programmer[4] I have my own sense of what I mean by abstraction and elegance in software. I often use the analogy of climbing a mountain. To realize that there is an abstraction that incorporates a number of previously disparate elements of one's code as special cases is like reaching a spot from which one can look out over the valley below. One has a better perspective of how things that were once familiar in detail are related to each other more abstractly. One also sees that one is not at the top, that another refactoring will take one to the next lookout area.

After climbing to that next overview one has an even broader perspective. But once one has understood the view from there one sees that with a bit more effort, one can reach a still higher level. It seems never to end. At each level one catches glimpses of yet higher peaks with better views. No matter how God-like one's view of the code, there seems to be an even more elegant and more abstract way of understanding it.

But I also find myself getting closer and closer my code. The analogy I use in describing that experience is to someone who loves woodworking. After building an initial version of an artifact she goes over it with fine sandpaper. She massages and caresses it until it's soft and smooth. As she works the material, smoothing it, polishing it, and shining it, it begins to glow with an inner light. Its tactile beauty is irresistible. One cannot refrain from running the fingers of one's mind over its subtly curved surfaces.

ROA'08, May 11, 2008, Leipzig, Germany.

When writing software one takes both journeys simultaneously. As the code becomes more abstract it also becomes more clarified, its structure more elegant. The farther the software as a vehicle for generality and abstraction allows one to see, the more one appreciates its detailed beauty, the ingenious simplicity of each element, the mind-satisfying shape of each line of code, the clarity of thought revealed by each construct.

One of the earliest proposed "methodologies" for software development was called "stepwise refinement." [12] Refinement in Wirth's sense meant decomposition from higher level constructs into lower level details. That sort of refinement is not what I find myself doing when I develop software. But if one understands *refinement* in its other sense—to polish, to improve, and to perfect—then for those who appreciate it, software development is indeed successive (if not necessarily "stepwise") refinement.

3. REFACTORING: THE ROYAL ROAD TO ABSTRACTION

Refactoring is commonly understood as "the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure." [6]

Why would anyone want to do that? If refactored software behaves no differently from unfactored software, why bother? The standard answer appeals to pragmatic benefits: when one later wants to modify the software the refactored code with its "improved internal structure" will be easier to understand and less expensive to fix or extend.

True as that may be I doubt that most programmers would agree that the ultimate measure of software quality is how easy it is to maintain. Yet, most expert programmers are fanatical about refactoring. Why? Expert programmers refactor to reveal and express abstractions—in particular the abstractions that identify and reflect the essential structure of the software. Plato[5] spoke of "carving Nature at its joints." When we refactor software, we don't so much as carve the software at its joints as bring those joints into focus, like seeing the software in an X-ray picture. What had seemed somewhat amorphous comes to have bones and other essential skeletal structures. Refactoring is the identification of an underlying structure.

Here's an example. In writing an expression evaluator in Prolog[6] one could write the following.[7]

```
eval(A + B, C) :-
   eval(A, AVal),
   eval(B, BVal),
   C is AVal + BVal.
```

---

[4] My Ph.D. is in Computer Science, and I am a professor of computer science. But at heart I'm a programmer. The job title of *programmer* is commonly denigrated as referring to a low level task. I think that's a mistake. One of the most important attributes of software is that it's executable. That requirement is what keeps software developers honest. To create executable software is to program a computer.

[5] *Phaedrus* 265d-266a.

[6] If you feel uncomfortable reading Prolog code, see Section 5.

[7] Prolog programs manipulate tree structures. Thus **A + B** is the tree structure with root **+** and children **A** and **B**. The Prolog **is** operator evaluates its right hand argument (as long as it is an arithmetic expression) and attempts to unify the result with its left hand argument. Thus the first clause (the first four lines) says that to evaluate the expression **A + B** one should evaluate **A** and **B** individually and then apply **+** to the results.

```
eval(A - B, C) :-
   eval(A, AVal),
   eval(B, BVal),
   C is AVal - BVal.
eval(A * B, C) :-
   eval(A, AVal),
   eval(B, BVal),
   C is AVal * BVal.
…
```

But one could also write the following.

```
eval(Expr, Result) :-
   Expr =.. [Op | Args],
   isArithmeticOp(Op),
   map(eval, Args, ArgsVals),
   Expr1 =.. [Op | ArgsVals],
   Result is Expr1.
```

Why is this better? There is far less code. More importantly, it is more powerful because it is explicit about what one does to evaluate an arithmetic expression in general. One extracts the arguments, evaluates them, reconstitutes the expression using the argument values instead of the arguments, and then performs the expression's operator on the result.[8]

In being explicit about how to evaluate arithmetic expressions this refactored version allows one to apply that process to any arithmetic expression.

Of course this is not new; it is procedural abstraction, the sort of abstraction represented by any software subprogram. This example may seem more elegant because it manipulates symbolic expressions. But it's really no different.

The conversion of the first version of the program to the second is a prototypical example of refactoring. The process of evaluating arithmetic expressions has been made explicit and reified by *factoring it out* of the original code and expressing it separately.[9]

By extracting and reifying the essence of the expression evaluation process one has created a unit of knowledge. This is not a minor issue. Virtually every concept is an abstraction. When we as human beings create and name a concept we have captured and domesticated it. From then on we can put that concept to work whenever we need it. That's the power of abstraction.

4. CLIMBING MOUNT ABSTRACTION

This section illustrates how one abstraction can lead to another.

Figures 1 and 2 show standard Java versions of depth-first and bread-first search.

```
public Node dfSearch(Node node) {
   if (this.isSatisfiedBy(node)) return node;
   for (Node child : node.children) {
     Node solution = this.dfSearch(child);
     if (solution != null) return solution;
   }
   return null;
 }
```

Figure 1. Depth-first search.

```
public Node bfSearch(List< Node > frontier) {
   while (!frontier.isEmpty()) {
     Node node = frontier.remove(0);
     if (this.isSatisfiedBy(node)) return node;
     frontier = append(frontier, node.children);
   }
   return null;
 }
```

Figure 2. Breadth-first search.

The idea is that these searches are methods in a Goal class. A goal (instance of Goal) is passed the root Node of a tree, which is searched for a node that satisfies it. The line

```
this.isSatisfiedBy(node).
```

tests whether node satisfies the goal (i.e., **this**).

The method dfSearch uses recursion to perform a depth-first search. The method bfSearch uses a frontier list (of Nodes), which stores the nodes to be examined at the current and next levels, to do a bread-first search. When frontier is passed to bfSearch it contains the tree's root node as its sole element.

Although it may appear that dfSearch does not keep a list of nodes to be searched, in fact it does, in its call stack. If the implicit list embodied by the depth-first search call stack is made explicit—let's call it frontier also—the two search approaches can be integrated as follows.

---

[8] The Prolog operator **=..** converts between its left hand argument, which is a tree, and its right hand argument, which is a list whose first element is the root of the tree, and whose remainder is the list of children of the root. The Prolog notation for a list is **[Head | Tail]**.

[9] It is appropriate that the term *abstraction* should be used for a process that extracts the essence from something. The Online Etymological Dictionary [8] offers this entry for *abstract*.

> 1387, from the Latin *abstractus* "drawn away," past participle of *abstrahere*, from *ab*(s)- "away" + *trahere* "draw." Meaning "withdrawn or separated from material objects or practical matters" is from 1557.

The original late fourteenth century sense of *abstract* was to draw away from. One still finds that sense in current dictionaries: to abstract is to extract. By the mid sixteenth century, a sense evolved that the act of drawing away separated the essence of something from its material realization.

```
public Node parameterizedSearch(List< Node > frontier, boolean depthFirst) {
  while (!frontier.isEmpty()) {
    Node node = frontier.remove(0);
    if (this.isSatisfiedBy(node)) return node;

      // For depth-first search put the children at the front of the frontier, simulating a stack.
      // For breadth-first search put the children at the end of the frontier as usual.
    frontier = depthFirst   ? prepend(node.children, frontier)
                            : append(frontier, node.children);
  }
  return null;
}
```

Figure 3. The search type has been parameterized: depth-first or breadth-first.

```
public Node abstractedSearch(List< Node > frontier, SearchType searchType) {
  while (!frontier.isEmpty()) {
    Node node = frontier.remove(0);
    if (this.isSatisfiedBy(node)) return node;

      // searchType.merge(frontier, node.children) merges node.children
      // into frontier. Depending on the SearchType, the merge may yield
      // breadth-first, depth-first, best-first, A* or some other search strategy.
    frontier = searchType.merge(frontier, node.children);
  }
  return null;
}
```

Figure 4. The search type has been abstracted.

(a) Both searches keep a list of nodes to be examined called a *frontier*. (b) In both searches, nodes are removed from the front of the frontier. (c.1) In breadth-first search the frontier is treated as a queue (the list of children is appended to the frontier). (c.2) In depth-first search the frontier is treated as a stack (the list of children is prepended to the frontier). See Figure 3.

One may argue, justifiably, that combing breadth-first and depth-first search does no honor to abstraction. After all, these two searches have very different properties. Depth-first search is not complete whereas breadth-first is, and breadth-first search is much more memory intensive than depth-first. Figure 3 makes the two seem almost like twins, which they aren't.

Nonetheless, Figure 3 illustrates two important features of refactoring. (a) An implicit element (the depth-first search list of yet-to-be-examined nodes) is made explicit. (b) An underlying procedural similarity is extracted, reified, and parameterized.

More interesting is the next step. Figure 3 suggests that the key to search types is how the children of a node are added to the frontier—at the front or at the back. Now consider best-first search, which differs from breadth-first and depth-first in that its frontier is sorted by a measure of goodness. To add best-first search to Figure 3 one would merge the children into the frontier according to that measure of goodness.

But why not abstract out the entire question of how to combine the children and the frontier? Figure 4 illustrates how to do that: pass to the search method an object that represents the search type and let that object decide how to combine the children and the frontier. This refactoring reifies the search type itself.

The preceding two refactoring steps illustrate the transformation of an implicit computation element (first a list and then a code segment) into a first class computational element. They also illustrate how one abstraction can serve as a basis for another as one climbs the mountain of abstraction.

One could continue by noting that the frontier need not be a list. All that is required is that the frontier (a) serve up elements if it has any and (b) accept new elements and determine in which order to serve them. One could create a Stream class that provides that service. It would be parameterized by an ordering component, which in this case would be a SearchType object.

The fundamental design pattern. Instead of continuing along the lines of the previous paragraphs, let's step back from code level abstractions and abstract the second abstraction step itself. Notice that virtually any **switch**-like computation is potentially abstractable. Consider the following code fragment.

```
switch (expression) {
  case 1: code_1;
  case 2: code_2;
  …
}
```

Instead of the preceding (or an equivalent as in the conditional expression in Figure 3) one can create a class that represents the generic activity of the cases with a subclass for each case.

Abstracting a collection of comparable behaviors in this way is known as the Strategy Design Pattern [7]. The strategy design pattern is in my view the fundamental abstraction: the reification of a distinction into a conceptualization. As discussed in section 1, the difference between a distinction (like a **switch** statement) and an abstraction is the explicit identification and reification of the distinction (as in a class). A distinction becomes an abstraction

when one reifies and names the category or concept that embodies or carries the distinction.[10]

Although the Strategy Design Pattern is fundamental, all design patterns are abstractions. Every design pattern identifies and reifies a generic arrangement of software elements.

5. WHERE TOP-DOWN MEETS BOTTOM-UP

The title of the preceding section deliberately echoes Dawkins' *Climbing Mount Improbable* [3], which explains how evolution creates improbably intricate designs through random variation and environmental selection. Although I agree with [4] that all creativity, even human creativity, is a result of random variation and selection, that's not why I echoed Dawkins' book title. I echoed Dawkin's book title to make the point that evolution and abstraction are both bottom-up processes. Both build higher levels (of functionality or abstraction) from an existing base.

Evolution is said to be a blind tinkerer. Its products are bricolage, cobbled together from whatever happens to be available—a bottom-up process.

As Section 4 illustrated, abstraction too is a bottom-up process: new abstractions build on previous abstractions.[11] Also, since abstractions result from the extraction and reification of the essence of something one cannot extract an essence from something until that thing exists.

Abstraction as self-awareness. One of our gifts as human beings is a degree of self-awareness. Like abstractions that build on each other at the code level, abstractions that grow from self-awareness are also bottom-up.

A significant thread in computer science can be understood as self-awareness. The process of *becoming aware of and then reifying a thought process as an abstraction* can be seen from the earliest days of computing. John Backus conceived Fortran when he realized that it was possible to program a computer to do the same sort of translation that programmers did mentally.

Software developers soon abstracted the compilation process itself by defining grammars for programming languages and building them into compilers. The next level of abstraction resulted from the development of compiler generators, which reified grammars as computational objects. Virtually all programming languages, tools, and frameworks reflect a similar trajectory of progressive self-awareness and reification. Abstraction as self-awaraeness is continued in Section 6.2.

Why bother making this point? There seems to be a never-ending debate about how software is developed: top-down, bottom-up, or "middle-out." As just discussed, abstraction in software development is bottom-up—although see section 7, requirements analysis, where abstraction precedes software.

But consider software composition itself, the activity of writing a unit of software. In writing a subprogram, for example, one organizes control structures, expressions, statements, other subprograms, etc. In writing a class one organizes instance (and perhaps class) variables, methods, etc. Composition involves (a) organizing software elements (b) to serve a given purpose.

Is this a top-down or bottom-up activity? One composes a software unit to embody a meaning. Since one has a meaning in mind before writing the unit, then with the usual qualifications about iteration this seems to be a top-down activity. At the same time, one composes the unit by arranging existing elements. So it also seems to be bottom-up.

One may argue that in writing a software unit one often composes it of elements that don't yet exist—a subprogram may call other subprograms that haven't been written; a class may include methods and refer to other classes that don't yet exist. So composition must be a top-down process—as in Wirth's original notion of stepwise refinement.

I think that's a misleading analysis. When one writes a unit of software it's not relevant whether the components already exist or that one simply believes that they can be written. The components of meaning exist—if only in one's mind. Whether or not a particular unit of meaning has already been fabricated as software is secondary. Composition is necessarily a bottom-up process. How could it not be? To compose is to arrange elements into some organized structure.

Software composition is thought externalization—discussed further in section 7. Thought externalization of any kind is the act of recording one's thoughts in symbolic form. The thought must exist before it can be externalized; the elements used to represent the thought must exist before they can be put together to express the thought. Thus to externalize a thought, one brings together the thought (top-down) and the means to express it (bottom-up), both of which must exist to externalize the thought. To compose an external representation of a thought—in software or in natural language—one must understand how to combine meaning units to express the thought. Top-down meets bottom-up when one formulates a thought for externalization.

6. CAN ABSTRACTION BE TAUGHT?

I have three answers: yes, yes but with difficulty, and no.

6.1 Yes, abstraction can be taught

We teach it every day. When we teach a programming language we are teaching abstraction. When we teach design patterns we are teaching abstraction. When we teach an architecture framework, we are teaching abstraction. In these and similar cases we are teaching students to use pre-packaged abstractions — abstractions that most of them would not invent themselves.

Of course some students will use pre-packaged abstractions more skillfully than others. But that's true of every skill. The point,

---

[10] Just because one can build such an abstraction doesn't mean that one must. The Prolog example discussed above converted from a number of special cases (one for each arithmetic operator) to a more integrated approach. But the integrated approach was not organized as a general case along with special (subclass) cases for each arithmetic operator. The integrated version refactored out the common supporting and peripheral processing (evaluation of the expression subcomponents) and then left it to the Prolog **is** operator to evaluate the main arithmetic operator. Prolog's built-in **is** operator handles evaluation of all arithmetic operators. Its implementation may use a **switch**-like construct, but that's at a level below the scope of this example.

[11] James Burke has made a career out of tracing such sequences. His *Connections*, a popular public television series three decades ago, was accompanied by an eponymous book [2].

though, is that once an abstraction has been identified and captured in some recorded formulation, it can be taught. I would be willing to bet that because our knowledge of software abstraction continues to advance, today's programmers produce significantly better software—with more effective use of abstractions—than that produced previously.

Improvement of this sort illustrates the power of culture. Culture enables us to record and share knowledge. We don't have to discover for ourselves everything that isn't built into our genes.

Interestingly, some computer scientists think that using pre-packaged abstractions is a bad idea. Dewar and Schonberg [5] discuss what they call a "worrisome trend."

> The irresistible beauty of programming consists in the reduction of complex formal processes to a very small set of primitive operations. Java, instead of exposing this beauty, encourages the programmer to approach problem-solving like a plumber in a hardware store: by rummaging through a multitude of drawers (i.e. packages) we will end up finding some gadget (i.e. class) that does roughly what we want. How it does it is not interesting!

Although I wouldn't use such a dismissive term, teaching students to "rummage" about until they find the right pre-packaged abstraction is exactly the right approach. (And one shouldn't care how the abstraction is implemented as long as it meets one's time, space, and scalability needs.) One shouldn't ask students to reduce all programming to a very small set of primitive operations. Doing so would be to tell them to throw away the abstractions our community has developed. Students should be taught to appreciate how rich a treasure trove of abstractions is available. A significant aspect of the production of elegant code is finding the right abstractions.

But Dewar and Schonberg aside, even if we agree that teaching students about pre-packaged abstractions is a good idea, there is a serious qualification to the assertion that such abstractions can be taught. To understand many abstractions one must first understand notations and prerequisite abstractions.

I've already discussed how abstractions build upon previously developed abstractions. Notations present a special problem. Notation is generally easy to teach, but it is often difficult to learn a notation well enough that it becomes second nature. Notation shouldn't stand in the way between a person and an idea. Yet it frequently does.

When most of us read something in our native language, the sentences generally enter our minds more or less directly as meaning. We don't struggle first to translate the words into individual units of meaning before combining them into meaningful sentences—as we would with a language we don't know well. Yet that's what happens with unfamiliar notation.

Think of the difference between looking at code in a programming language you know and code in a programming language you don't know. How comfortable were you, for example, with the Prolog code earlier in this article?

I know Java reasonably well; I've been programming in it since it was first introduced. When I see a page of Java code I feel at home. I've never felt that way about C or C++. It's even worse now; I haven't written any C/C++ code in a long time. When I look at a page of C/C++ my mind rebels. What are those strange ampersands, asterisks, and arrows? I don't want to have to translate them into meaning before I can understand the code.

Do you know Haskell? Consider the following Haskell code.

```
primes = sieve [2 .. ]
  where sieve (p:xs) = p:sieve [x | x <- xs, (x `mod` p) /= 0]
```

Does it feel comfortable? It's a definition of the prime numbers. It's beautifully elegant. But it relies on many concepts and notations—including the idea of infinite lists and a notation for list comprehension and list construction. In addition, it puts mod in strange-looking single forward-quotes. One can probably guess what **where** means, but it's not something one sees in most languages. It appears that sieve is defined recursively. But where is the termination condition? With Haskell's infinite lists and lazy evaluation, one doesn't need a termination condition.

Now consider this widely quoted Haskell formulation for the Fibonacci numbers.

```
fib = 0 : 1  : zipWith (+) fib (tail fib)
```

Again, quite elegant. It too uses infinite lists and lazy evaluation. It uses the standard library function tail (whose meaning one might easily guess) and the less transparent zipWith, which apparently takes (+), whatever that means, as an argument. If you didn't know Haskell would you know that the second ":" creates a list whose tail is the list returned by zipWith?

This definition also seems to be recursive—and again with no termination condition. Not only that, the recursive reference seems to define fib directly in terms of itself: the two fibs that appears in the body aren't applied to a modified version of the arguments. At least in primes, sieve was applied to something derived from the tail of the argument in the sieve signature. But then fib doesn't have any arguments. So what is fib, a function with no arguments, a list? How should all this be understood?

The answer is that fib is a list, not a function, and it's *not* defined recursively. This fib definition works by constraining the list fib to be the Fibonacci numbers. The first two are given. From the third element on each Fibonacci number is the sum of the previous two. How is that accomplished? If one pair-wise adds the list of Fibonacci numbers (fib) to the tail of the list of Fibonacci numbers (tail fib), one gets the list of Fibonacci numbers starting with the third Fibonacci number. That's what zipWith (+) does. Once that structure is set up, the list of Fibonacci numbers can be generated iteratively (and lazily), not recursively.

The point of these examples is not to criticize Haskell, which is a wonderful language. It is that when abstractions are expressed in terms of unfamiliar prerequisite abstractions and notations it is sometimes with difficulty that they precipitate as meaning in one's mind. This is a serious problem. It often takes time for ideas and notations to become familiar enough to serve as a foundation for further ideas. Frequently it's not new ideas that are difficult to learn; it's the language in which new ideas are expressed that creates a barrier to understanding.

6.2 Yes, self-awareness can be taught, but with difficulty

Self-awareness is a far different skill from applying pre-packaged abstractions. As discussed above, by *self-awareness* I am referring to an awareness of one's own thought processes along with the insight that those thought processes can be captured, conceptual-

ized, and named—and when applied to software, externalized as code. Backus's genius was not the invention of Fortran. His genius was his awareness that the mental process of translating equations into machine code could be reified and automated. It is self-awareness at that level that leads to the most important software abstractions.

Even though self-awareness can be taught—at least I believe it can—it is not a skill that we generally attempt to teach. Self-awareness is more subtle than most skills. It is more difficult to grade, and its visible products are harder to measure. Because we have not directed our attention to teaching self-awareness, we have virtually no academic experience doing so.[12] Self-awareness belongs to no single discipline and has no natural academic home. No accredited University offers a self-awareness major.

But I believe that we can teach self-awareness and that we can do so as a concrete, discipline-specific skill. To teach an introductory course in software-oriented self-awareness we could give students code to which design patterns apply and ask them to improve the code. We would teach such a course *before* teaching design patterns—because in this course we are encouraging students to invent their own design patterns.[13]

Such a course would be difficult to teach. Many students will have difficulty inventing the needed design patterns and might find the course quite frustrating. For some students, once they discover that they are supposed to invent design patterns they will have great difficulty not looking up design patterns in advance and seeing which ones fit. Two points should be made to any student who does that. The first is that using a catalog of design patterns makes it impossible to practice self-awareness. So the student will be losing the benefit of the course. But the second is that the student has already practiced self-awareness at least once. In realizing that the objective is to make up one's own design patterns, the student has become aware of something on his own—and should be congratulated for doing so.

The student who truly misses out is the one who finds out about using a catalog of design patterns from another student. True, self-awareness helped even that student learn to ask another student, and that trick can be used over and over. But using it precludes experiencing self-awareness in other contexts.

The point of the course would be to help students learn to become aware of their own thought processes so that they could use that skill elsewhere—not for them to invent the right design pattern for each example. Could we teach this sort of self-awareness? I don't know. Do we have time to teach it? I don't know that either. There are so many known and useful abstractions that we spend all our time teaching them. Difficult as it may be, such a course would be worth the effort.[14]

[12] Debora Shuger, Professor of English, claims (personal communication) that the humanities teach it. I'm skeptical.

[13] Shuger claims that the humanities teach self-awareness when students perform the comparable activity with literature.

[14] The closest I can think of to something along these lines is the cataloging of what design pattern aficionados call *bad smalls*. But students who find and use a "bad smells" catalog are not finding and inventing abstractions on their own either.

6.3 No, abstraction as brilliance cannot be taught

Creativity and insight are not evenly distributed. It's difficult to imagine that one can teach anyone and everyone to be a brilliant programmer—or a brilliant composer, writer, or painter. Even though for most fields one can teach how to recognize elegance and creativity, I can't imagine that one can teach—as a dependable skill—how to produce elegance and creativity. I might be able to analyze one of Garrison Keiler's "Guy Noir" scripts and tell you why it's funny. But that doesn't mean I can write one. Perhaps this is another (unprovable?) example of $P \neq NP$. Production just seems to be a lot harder than recognition.

7. SOFTWARE IS EXTERNALIZED THOUGHT

So far I have focused on abstraction as a way of writing better software. This section looks at abstraction as a way of deciding what software to write.

In [1] I claim that computation is externalized thought and that software is the medium we use for that externalization. Although like most creative work, software development is usually an iterative process—write a version, see how it looks, modify it, etc.—to a first approximation, one can think of software as an external representation of an idea in the mind of the developer. When we write software we construct an idea of what we want the software to do and then externalize that idea as code.

The power of computing springs from the fact that we have developed languages that have both of the following properties.

- We can use the language to express our thoughts.
- The language is executable. It can have an effect in the material world without human intervention.

Never in history have we had such a capability. Every previous symbolic medium for thought externalization—except perhaps music—required a human being to interpret it. One underappreciated aspect of the computer revolution is the development of languages for externalizing thought.

That software corresponds to and represents ideas is not new. It is standard to refer to the conceptual model implemented by software. Unfortunately, we often treat conceptual models as if they were software by-products. Of course, the ordering is the reverse; the idea precedes the software. Software that doesn't match its conceptual model is said to have a bug—or as the joke goes, we change the conceptual model to make the bug a feature.

So if software is an externalized conceptual model, to be better software developers (and to train better software developers) we should improve our ability to create conceptual models. To do that we should study how we think and how we externalize our thoughts as software—and we should do so for as wide a variety of disciplines as possible. In other words, we should do our best to understand how people use computers to help themselves think.

To do that, it makes sense for students to experience a wide range of domain-specific applications—especially those with powerful visualization and data manipulation features, i.e., those which externalize thought most visibly.

In helping students learn about the abstractions used in different disciplines it's important to focus concretely on how a discipline

uses its abstractions to model the world—and not on the abstractions themselves as the subject matter. I call this the model-first approach in contrast to the theory-first approach.

A simple example of the distinction between model-first and theory-first is that if we are teaching about a discipline that uses numbers to count (and I mean something as simple and basic as counting), we should not require that students first master number theory. One need not know what Goldbach's conjecture is before one can use numbers to count.

Obvious as this may seem, we don't always follow this rule. A course that teaches students how to build systems dynamics models might use a simulation engine based on differential equations. Such a course should *not* attempt to teach the mathematics of differential equations. The abstractions needed to build models based on rates of change are not very difficult; the mathematics of differential equations is more sophisticated.

Appropriately, we use the model-first approach in our own discipline. We teach beginning computer science students how to build mental models and how to externalize those models in a programming language. We don't start by teaching students formal language theory, compilers, and techniques for implementing operational semantics in virtual machines.

To pursue a model-first approach we should teach students as many ways as possible to build mental models of the world and to externalize those models as software. Students who understand how to develop mental models—and especially mental models that can be externalized as software—will always be in demand. They will serve as knowledge middlemen—otherwise known as requirements analysts.

8. SUMMARY

The subject matter of this workshop is abstraction in software engineering. Abstractions are concepts. Software is nothing if it is not about concepts. Abstraction plays two important roles in software engineering. (a) We use abstraction to build more powerful and more elegant software. The richness, continuing growth of our software abstraction ecology, and the vibrancy of our software community suggests that we have not come close to exhausting the space of software abstractions. (b) As human beings, we use abstraction to build mental models of our world. Software engineers help with both (i) the development of these models and (ii) their externalization and animation as software.

9. REFERENCES

All Internet accesses are as of January 24, 2007.

[1] Abbott, Russ, "If a tree casts a shadow is it telling the time?" *International Journal of Unconventional Computation*, Vol. 5, No. 1, pp. 1–28, 2008 (to appear). Preprint: http://cs.calstatela.edu/wiki/images/6/66/If_a_tree_casts_a_shadow_is_it_telling_the_time.pdf.

[2] Burke, James, *Connections*, Little Brown & Company, 1978.

[3] Dawkins, Richard, *Climbing Mount Improbable*, Norton, 1996.

[4] Dennett, Daniel, *Darwin's Dangerous Idea*, Simon & Shuster, 1995.

[5] Dewar, Robert B. K. and Edmond Schonberg, "Computer Science Education: Where Are the Software Engineers of Tomorrow?" *Crosstalk: the Journal of Defense Software Engineering*, January 2008. http://www.stsc.hill.af.mil/crosstalk/2008/01/0801DewarSchonberg.html.

[6] Fowler, Martin, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.

[7] Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] Harper, Douglas (ed), *Online Etymological Dictionary*, http://www.etymonline.com/index.php.

[9] Knuth, Donald E. "Computer programming as an art,." *Communications of the. ACM* 17:667-73, 1974. http://fresh.homeunix.net/~luke/misc/knuth-turingaward.pdf.

[10] Kramer, Jeff and Orit Hazzan, "The Role of Abstraction in Software Engineering," in *Proceeding of the 28th international conference on software engineering*, ACM, 2006.

[11] Liskov, Barbara, *Program development in Java.* Addison-Wesley, 2000.

[12] Wirth, Niklaus, "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, No. 4, pp. 221-227, April 1971. http://www.cs.unca.edu/~brownsmi/0408_Fall/csci431/resources/generalReference/PgmDvmtByIterative.htm.