# Beyond AOP: Toward Naturalistic Programming

Cristina Videira Lopes, Paul Dourish
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697
+1-949-824-1525/8127
{lopes, jpd} @ ics.uci.edu

David H. Lorenz, Karl Lieberherr
College of Computer and Information Science
Northeastern University
Boston, MA 02115
+1-617-373-2076/2077
{lorenz, lieber} @ ccs.neu.edu

## ABSTRACT
Software understanding for documentation, maintenance or evolution is one of the longest-standing problems in Computer Science. The use of "high-level" programming paradigms and object-oriented languages helps, but fundamentally remains far from solving the problem. Most programming languages and systems have fallen prey to the assumption that they are supposed to capture idealized models of computation inspired by deceptively simple metaphors such as objects and mathematical functions. Aspect-oriented programming languages have made a significant breakthrough by noticing that, in many situations, humans think and describe in crosscutting terms. In this paper we suggest that the next breakthrough would require looking even closer to the way humans have been thinking and describing complex systems for thousand of years using natural languages. While natural languages themselves are not appropriate for programming, they contain a number of elements that make descriptions concise, effective and understandable. In particular, natural languages referentiality is a key factor in supporting powerful program organizations that can be easier understood by humans.

## Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *control structures.*

D.3.2 **[Programming Languages]**: Language Classifications – *Very high-level languages.*

## General Terms
Documentation, Design, Human Factors, Languages.

## Keywords
Language Design, Aspect-Oriented Programming, Natural Language, Cognitive Foundations of Programming.


OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

## 1. INTRODUCTION
Software systems in fields such as Science, Engineering and Business are naturally complex, in that the structures and processes that they capture have a certain level of domain-specific complexity requiring domain expertise. But, on top of that natural complexity, the problem of software development is worsened by the existing software development technologies. The kernel of the problem is that the existing description mechanisms – programming languages – are inadequate when it comes to conveying relevant information to people about the software systems. The established programming paradigms fail in providing appropriate support for non-hierarchical concerns, additional or custom-made behavior, and non-trivial domain-specific structures and processes. As a result, software systems become fragile "monuments" of code that only illuminated software "artisans" (i.e. projects leads) dare to modify, preventing the real domain experts to control the software in a direct and systematic manner.

The use of architectural descriptions, rigorous software development practices and modern programming languages and tools helps, but fundamentally remains far from solving the problem. The kernel of the problem is the lack of support for program understanding by the different people involved in the project. Multithreading, exceptional cases, optimizations, and the like, contribute to the natural complexity of the programs. But a considerable part of the complexity is due to the fact that programmers, when writing the code, are forced by the programming language to write it down in arcane and esoteric ways that are a long way from expressing the natural intentions behind the code. As a consequence, many times programmers prefer to write blocks of code from scratch rather than having to understand and debug other people's code. Especially when working at the systems level, code that communicates effectively to the machine rarely communicates effectively to human readers.

### 1.1 Programming Reflects Thinking
Researchers are constantly looking for ways to express the programs in a form that more closely follows the way programmers think before they are forced to break their thoughts in operational details imposed by the existing programming languages. We know that this is possible, because when programmers are asked to explain their code, they do so concisely, skipping operational details, sometimes using a thought flow that is quite different from the control flow in the code. Our goal is to address the gap between those two forms of explanation.

Over the years, several languages, both textual and visual, have been designed that focus specifically on issues of usability and expressiveness (e.g., [60] and [10]). They usually target children, the novice programmer, end-user programming and rapid prototyping. None of these languages, however, has had much success in systems software development. The problem is that those languages and environments present simplified models of computation that cannot support the demands of systems programming.

In the domain of professional programming, Object-Oriented Programming and Aspect-Oriented Programming (AOP) [20] have been addressing some related issues within existing programming languages, targeting complex software applications. In particular, AspectJ [1], an AOP extension to Java, allows programmers to localize crosscutting concerns such as tracing, logging or profiling in program modules of their own and outside the classes. That way, code that reflects some important design units, but that Java forces to be spread throughout the system, can be encapsulated in their own modules, improving readability, maintainability and configurability. The success of AspectJ is due, in part, to the acknowledgement in the software industry that the problems of programming language expressiveness are serious; AspectJ embodies an approach which makes code more expressive, more readable, and more reliable, and can address industry's need for improved reliability and decreased development time. AspectJ achieves this by making programs follow more closely the intentions of their developers. But even AspectJ is still far from supporting the natural expression forms we are looking for.

## 1.2 The Role of Natural Languages

Before computers came along, people were successfully defining and disseminating complex systems for thousands of years, using a technology they come equipped with: natural language. They were writing all sorts of documents containing structure and process information, ranging from specification manuals of complex systems to constitutions of social organizations. Writing a structured document, for example, the Constitution of the State of California, requires much more than simply putting words together in grammatically correct sentences; it requires dividing the subject matter into smaller and smaller units, e.g., chapters, sections, subsections, paragraphs, sentences, that convey semantic information to the reader. Those small units don't exist in isolation; they refer to and use each other. They do so in ways that obey the rules of the natural language and that very rarely obey the simple functional module structures supported by existing programming languages.

Computer programs, of course, are different. They must define structure and process for computational systems, and therefore must address issues of data types and control flow. Systems software development is hard partially because of the inherent complexity of the structures and processes that they convey. But a considerable part of the burden of systems software development is due to the complexity added by the operational details imposed by programming languages, such as having to deal with temporary variables or having to cope with exceptional cases. We argue that such problems are historical artifacts. They reflect the legacy of traditional programming languages in either machine languages (resulting in an overriding concern with control flow and assignment) or mathematical formalisms (resulting in an overriding concern with binding and transformation.) Existing programming languages force programmers to express ideas using a narrow support for structural and reflective referencing and a total lack of support for temporal referencing.

## 1.3 Contribution

The main purpose of this paper is to re-generate some discussion around the role of Natural Languages in Programming Language design. We are aware that this is a relatively old theme that can be traced to the 1960's [55]. A lot has happened in Linguistics and in Programming Languages since then. In particular, some lessons learned from Aspect-Oriented Programming lead us to believe that there is value in revisiting the issue now. AOP made us pay more attention to the way certain referencing mechanisms occurring in natural languages allow us to express some ideas that can't be easily expressed in traditional programming languages.

In this paper we suggest two new ways of thinking beyond AOP, the first one within the AOP framework, and the second outside that framework. We make the argument that the primitive abstractions in programming languages should be drawn from the study of Natural Languages, rather than from Computer Engineering or Mathematics or ad-hoc metaphors such as Objects.

The reminder of the paper is organized as follows. In Section 2 we revisit AOP, highlighting the English-equivalents of the expression mechanisms in some AOP languages. In Section 3 we state some improvements that can be done in AOP systems, while remaining centered in the AOP paradigm. Section 4 builds on the observations in previous sections and drafts a programming language way beyond AOP. Section 5 describes the relevant fields of research that should be taken into consideration. Finally, Section 6 concludes the paper in a rather inconclusive manner.

# 2. ASPECT-ORIENTED PROGRAMMING

Understanding the leap between object-oriented modular programming and aspect-oriented crosscutting programming is crucial to contemplate a leap beyond AOP. We have been deeply involved in the development of Aspect-Oriented Programming [20] and several flavors of it, D ([34], [35], [36]), AspectJ ([37], [21]), Demeter [32] and Aspectual Collaborations [31]. AspectJ, now reaching the 1.0 version, is a stable extension to Java used by large numbers of software engineers in industry.

The following is a brief historical overview of AOP that illustrates how it comes one step closer to certain referencing mechanisms in natural languages.

## 2.1 Domain Specific Languages: D

D was a domain-specific language that targeted two issues: synchronization of threads and parameter passing in remote method invocations. In this summary we describe only the synchronization issue. In D, coordinator modules, separated from Java classes, encapsulated the synchronization of threads. For example, the following coordinator module mandates the synchronization of `BoundedBuffer` objects:

```
coordinator BoundedBuffer {
    selfex put, take;
    mutex {put, take};
    condition empty = true, full = false;
    put: requires !full;
```

```
        on_exit {
          if (empty) empty = false;
          if (usedSlots == capacity)
        full = true;
        }
  take: requires !empty;
         on_exit {
           if (full) full = false;
           if (usedSlots == 0)
             empty = true;
         }
}
```

What this says, in English, is the following:

- This is a coordinator for `BoundedBuffer` objects.
- The operations put and take are self-exclusive, i.e., no two threads can execute either of them simultaneously.
- The operations put and take are also mutually exclusive, i.e., no two threads can execute both of them simultaneously.
- Let's define the conditions `empty`, which should be true in the beginning, and `full`, which should be false in the beginning.
- For the operation put: its execution requires that `full` is false; after exiting the operation: if `empty` was true, then the buffer is not empty anymore; if, on the other hand, the buffer reached its capacity after this put operation, then now it's full.
- <similar for take>

The binding between coordinator code and object code is done by the name of types, such as `BoundedBuffer`, and operations, such as put and take. Coordinators could also directly refer to internal variables of the classes, illustrated in this case by `usedSlots`.

## 2.2 General Purpose Language: AspectJ

In AspectJ, this idea was expanded and generalized. AspectJ is a general-purpose aspect language that uses the concept of "join point." Join points in AspectJ are points in the execution (run-time) of a Java program that programmers can name and handle at program time. Examples of join points are the beginning of a certain method execution, the invocation of an operation on an object, etc. In AspectJ, aspect modules, separated from Java classes, can encapsulate not only synchronization but also a variety of crosscutting concerns such as debugging or notification. For example, the following aspect mandates the display update upon moving objects, involving different operations in objects of three different types:

```
aspect DisplayUpdating {
  pointcut move():
   call(void FigureElement.moveBy(int, int))
   || call(void Line.setP1(Point))
   || call(void Line.setP2(Point))
   || call(void Point.setX(int))
   || call(void Point.setY(int));
  after() returning: move() {
    Display.update();
  }
}
```

What this means, in English, is the following:

- This is an aspect called `DisplayUpdate`.
- First let's define a set of join points consisting of the invocation of: `moveBy` in `FigureElement` objects, `setP1` in `Line` objects, `setP2` in `Line` objects, `setX` in `Point` objects and `setY` in `Point` objects; let's call this set move.
- When the computation reaches any of the join points in move, and after returning from the invocations, perform `Display.update`.

The thesis behind AspectJ is that certain units of program specification or design—in this case, the display update upon objects that have moved—have a systemic nature that cuts across any of the single object modules that those units pertain to. This thesis seems to be meaningful for software engineers at large, who have been adopting AspectJ enthusiastically.

## 2.3 The Kernel of AOP

What is it about Aspects that makes them both attractive to researchers and useful to practitioners? Consider tracing, for example. When we think of tracing, we formulate something like this: "for all methods, call Trace.in before they start executing and Trace.out after they finish executing." However, all programming languages will force us to transform this sentence into something like this: "In method A, call Trace.in; … call Trace.out; return. In method B, etc." So what is it about the first representation of the intention that's better than the second, and how does the natural language help? In this case it's the references to "all methods," "before … executing" and "after … executing". That is the power of AspectJ: it supports a richer set of structural and temporal referencing that follows what we have in natural languages. AspectJ does it in a way that seems to be very useful for practitioners: it allows the encapsulation of these forms in modules that can be added to or removed from the applications with a compilation switch. In other words, writing a tracing aspect is like writing a different chapter, or section, in a book.

So, what makes an Aspect be an Aspect, before we even think of programming it with AspectJ? Given the name chosen for it, which clearly influences our perception, Aspects are software concerns that affect what happens in the Objects but that are more concise, intelligible and manageable when written as separate chapters of the imaginary book that describes the application. This pseudo-definition of Aspect aligns well with what users have been using AspectJ for. The structural and temporal referencing in AspectJ are essential mechanisms for achieving the separation between the Objects and those other concerns. Those mechanisms are also naturalistic: we would use those kinds of referential relations if we were to write it in English, Portuguese or Hebrew. But the need for better referencing mechanisms doesn't end with what the word "Aspect" conveys.

# 3. LESSONS FROM AOP

AspectJ is, by no means, the ultimate language and model that solves the program-understanding problem. A lot more can and needs to be done. But there are several important lessons to be learned from AspectJ and AOP that can feed into the next generation of language support for complex systems.

## 3.1 Binding between Aspects and Objects

Once the application objects and the aspect routine are un-tangled and decoupled, they may bind with various degrees. If the aspect routine has great relevance to the application objects, it may palpate every element in the program and potentially affect every heartbeat of its execution. If the aspect routine has no bearing on the application program whatsoever, there is no interaction. We call this the binding extent of putting an aspect routine and an application program together. In AspectJ, pointcut designators determine the binding extent and join points are the binding elements.

With the binding model in mind, we can begin to look at some more flexible AOP mechanisms. We identify three characteristics of the binding extent:

**Spread**: The binding spread is the size of the cut, i.e., the number of different join points the aspect binds to. A logging aspect may affect every method of the code. Therefore the binding between the logging aspect and any program is typically wide spread. In contrast, an advice that only introduces a variable to a particular class in a particular program, and only to that class, has a very narrow spread: a single join point. The binding spread is a metric over the crosscutting and tangling resolution.

**Form**: Aspects have various forms of interaction with objects. The binding form is the model of the join points. In AspectJ, the form is mainly event-based, the events being the underlying object execution events. In aspectual collaborations [31], the form is a collaboration-oriented join graph. Complex descriptions may need binding forms beyond that expressible within the existing join point models.

**Granularity**: The granularity of the binding is the density of the underlying grid of potential hooks for aspects in the application program to bind to. In AspectJ, granularity is a property of the join point model and is independent of the particular application program or a particular aspect subroutine, but this need not be the case in general. The granularity influences the lower bound on the form and the upper bound on the spread.

## 3.2 AOP, Reflection and Metaobject Protocols

AOP has a deep connection with work in computational reflection and metaobject protocols ([57], [19]). A reflective system provides a base language and (one or more) meta-languages that provide control over the base language's semantics and implementation. The meta-languages provide views of the computation that no one base language component could ever see, such as the entire execution stack, or all calls to objects of a given class. Thus, they crosscut the base level computation.

Expressiveness is a goal, for which reflection is one powerful tool. We have exploited this connection to great advantage in our previous work on AOP. Early on, when prototyping AOP systems, we often started by developing simple metaobject protocols for the component language, and then prototype imperative aspect programs using them. Later, once we had a good sense of what the aspect programs need to do, we developed more explicit aspect language support for them.

Existing programming languages force programmers to express ideas using a narrow support for structural and reflective referencing [39] and a total lack of support for temporal referencing. AOP languages offer a reflective architecture. Unlike core reflection, which is structural, the aspectual reflection [22] is temporal, namely occurrence of join points. Natural languages seem to posses more temporal referential forms beyond what AOP currently provides.

## 3.3 Anaphoric Relations

One of the main characteristics of natural languages, which distinguish them from most formal languages, is the use of a diversity of anaphoric relations. Anaphora is, essentially, referentiality between utterances. Pronouns are examples of context-dependent anaphora: this, that, it, her, which, etc. But referent expressions can be more than pronouns. Natural languages support a multiplicity of possible forms that can be used to identify a referent in a given sentence or among sentences. In general linguistic usage, anaphora refers to referential dependence regardless of morphological form and regardless of whether it is context-dependent or context-free. In other words, ordinary pronouns and even full noun phrases count as anaphora. For example, they can be: lists of nouns such as "The president, the cat, the resident and the hat"; constraints on nouns "colorless liquids"; etc. We are using the term anaphora in this very broad sense. In this sense, as explained in Section 2.3, AOP supports a simple form of temporal anaphora. It can be extended to support a richer set.

## 3.4 Domain-Specific AOP Languages

Although Domain-Specific Aspect Languages (DSAL) are not the focus of this paper, DSALs might be a better approach than a general purpose aspect language. The reason is that domain specific aspect languages can utilize a higher level join point model. We can use a domain-specific language that generates code that is the basis for a higher-level join point model. The join points are then advised by additional aspects. For further arguments for DSALs, see [56].

# 4. BEYOND AOP

Elements of natural language, especially those pertaining to referencing, can create programming languages that are both expressive and executable. The goal of this paper is to identify the binding mechanisms in natural languages that will enable the description and organization of programs in a more natural way.

## 4.1 Example

To have a more clear idea of which relations are useful and which aren't, and to illustrate the objectives of a naturalistic programming language, we present an example in three steps. First we show a piece of Java code; second we show the same program using English words – the purpose of this second form is to illustrate what we do NOT seek; finally, we show another version of the same program, this time written in form we target.

### *4.1.1 Extreme 1: Description using Java (version 1)*

In the Ubiquitous Computing project at UCI we are developing applications using several hardware and software platforms. The applications include, for example, short-range acoustic modems, Personal Area Network protocols and speech processing [38].

These applications involve low-level systems programming, they are computationally intensive and, therefore, require a solid grasp of data structures, optimizations and multi-thread programming. They are written in Java and C, and the code is to be shared and (re)used by many students.

Consider the following code, extracted from one of our acoustic modems:

```
/**
 * encodeStream converts a given stream of
 * bytes into sounds.
 * @param input the stream of bytes to
 * encode
 * @param output the stream of audio
 * samples representing the input
 */
static void encodeStream(InputStream in,
                         OutputStream out){
 int readindex = 0;
 byte[] buff=new byte[kBytesPerDuration];
 while (  (readindex = in.read(buff))
        == kBytesPerDuration) {
  out.write(Encoder.encodeDuration(buff));
 }
 if (readindex > 0) {
   for (int i=readindex;
        i < kBytesPerDuration;
        i++)
     buff[i] = 0;
   out.write(Encoder.encodeDuration(buff));
 }
}
```

### 4.1.2 *Extreme 2: English Sugar-Coat (version 2)*

Now consider the following description using English. This English sugarcoat represents the other extreme (and is not what we advocate.)

```
encodeStream service
Summary: it converts a given stream of bytes
into sounds
It requires the following:
 An InputStream object known as in; it
  is supposed to contain the stream of bytes
  to encode.
 An OutputStream object known as out; it
  will be filled with the stream of audio
  samples representing the input.
It returns nothing
It is implemented as follows:
 . Create an integer called readindex and
   initialize it to zero.
 . Create an array of kBytesPerDuration
   bytes called buff.
 . A loop begins:
   . Request the service read from in,
     with argument buff; set readindex to
     the return value of this service.
   . If readindex is equal to
     kBytesPerDuration, then
     . Request the service write from
       out; the argument to this service
       is the return value of
       . Request the service encodeDuration
         from Encoder, with argument buff.
  End of loop.
  . If readindex is greater than 0 then
    . Set to zero all positions of buff
      starting at readindex.
    . Request the service write from out;
      the argument to this service is the
      return value of
      . Request the service encodeDuration
        from Encoder, with argument buff.
```

This description follows a similar philosophy to that of Hypertalk [60] and NaturalJava [53]. It is not much more than syntactic sugar over the Java programming model and language. Not only doesn't it help in understanding the implementation, but it is likely to be even worse for understanding a complex application, because it's a lot more verbose than the Java program. It misses the point.

### 4.1.3 *Something Else: Focus on the "Natural" Way of Describing What We Want (version 3)*

Finally consider this third version.

```
/**
 * encodeStream converts a given stream of
 * bytes into sounds.
 * @param input the stream of bytes to
 * encode
 * @param output the stream of audio
 * samples representing the input
 */
encodeStream(InputStream in,
             OutputStream out) {
  while there is data in in:
    read the first N bytes from it;
    perform encodeDuration on those bytes
    and write the result into out.

  if, however, after reading the input,
    the number of bytes read is less than N,
    then, before continuing, patch the
    resulting byte array of size N with
    zeros.
```

Let's assume for a moment this language can be implemented as is. The reader will probably agree that this version is the one that most concisely describes the intent of the implementation. This text could probably be easily implemented. What's valuable in this version is that it not only reads like English, but, moreover, organizes the ideas in a "natural" way and without "distracting" elements. The next section will analyze these points.

## 4.2 Analysis of the Target Language

Let's analyze the program in version 3 and compare it to versions 1 and 2.

- Versions 1 and 2 dwell in details of handling temporary variables; the last version doesn't mention any variables.
  - Instead of buff, it uses the natural dynamic binding "those bytes," which, according to standard English, refers to the bytes mentioned in the previous sentence.
  - Readindex is made redundant. This is because it was only there in the first place to cope with the exceptional case of when the input stream returns less bytes than what we asked for.

- Version 3 makes use of a reflective element: "this last operation." We consider this to be a reflective element, because it exposes knowledge about the underlying execution of the program by mentioning "operation."
- Most importantly, version 3 uses a subtly different organization of ideas. Namely, it first states the normal cases (i.e., we get the number of bytes we ask for out of the input stream), and only afterwards states how to handle the special case (i.e., we get less than what we ask for). In this case, the binding of the special case sentence with the place in the computational process where the special case might occur is done with the expression "after reading the input stream."

The third point must be carefully analyzed, because it embodies what we think are the most novel contributions of this proposal that can transform for the better the way people express ideas in programming.

These sorts of bindings, called anaphora in linguistics, are pervasive and perfectly natural when people speak and write documents. They are also natural ways of thinking about computational processes. However, existing programming languages lack appropriate support for them.

Existing programming languages are based on the premise that each statement, expression or function is a little "black box" that relates to the rest of the program through an input-output interface. This premise is made very clear in functional programming languages that reduce everything, including other languages' constructs, to functions. As a consequence, programmers are forced to stream their intentions into a series of sequential steps aligned with this very narrow pipeline view of the world.

So in this case, in the first two versions of the encodeStream function, the test of whether the read of the input stream returned less than expected is stated immediately after performing the read operation. This splits an important semantic unit—the occurrence and handling of the special case—in two statements whose relation is loosely established by the variable readindex:

```
while (   (readindex = in.read(buff))
       == kBytesPerDuration) {
 out.write(Encoder.encodeDuration(buff));
}
if (readindex > 0) {
  for (int i=readindex;
       i < kBytesPerDuration;
       i++)
    buff[i] = 0;
  out.write(Encoder.encodeDuration(buff));
}
```

As a consequence of this split, the write operation is repeated twice in the program text, once in the loop and again in the conditional that follows it. This is typical in existing programs, and it's extremely bad from an evolution point of view: when a specification changes, programmers must find all these redundant places and fix them by hand.

In this case, this redundancy could be avoided by using a do-statement like this one:

```
do {
  if ( (readindex = input.read(buff))
      < kBytesPerDuration)
    if (readindex > 0)
      for (int i=readindex;
           i < kBytesPerDuration;
           i++)
        buff[i] = 0;

  out.write(Encoder.encodeDuration(buff));
} while (readindex == kBytesPerDuration);
```

But in this case, the test of the value of readindex happens twice in each iteration of the loop rather than once. Furthermore, this organization emphasizes the special case: because of all those tests in the beginning, we can hardly notice what the loop is actually supposed to do most of the times. This is also typical and also bad.

In this pipeline view of the world, there is no way of refining a statement or expression or function at a later point in the program text. Yet, this refinement happens pervasively in written discourse. The existing programming languages have a very shallow support for structural referencing and a complete lack of support for temporal referencing.

In version 3, the test is stated as another sentence outside the lexical scope of the loop where the read occurs. The binding expression is "after reading the input stream". We can evaluate this expression unambiguously, in that we immediately understand that this expression refers to a point in time that has been established in the previous sentence "read … from the input stream." A programming language processor can also evaluate this expression correctly, if we make it do it.

## 4.3 Placing this "Language" into Perspective

There are two aspects pertaining to referencing: what to refer to and how to refer to it. This is, in fact, one of the most basic design decisions of any programming language. Programming languages have been highly biased in this decision. Here are some examples of things that are referred to. In low-level assembly languages, the what consists of registers and memory cells; in functional languages, it consists of functions and variables; in OOP languages, it consists of objects (very well-defined entities with a precise form), fields, variables and, when inheritance is included, classes. In typed languages, types are also part of what can be referred to. The mechanisms to refer to things vary from the use of syntactic forms to the explicit application of binding functions.

In contrast, Natural Languages have a much less well-defined set of things that can be referred to. In fact, the best word to describe what we can refer to is thing, which can be just about anything. It can be the computer memory and registers, for example; or functions and variables; or OOP's objects and classes; or types. But it goes way beyond these. It can be sets of things; it can be points in time; it can be "the previous paragraph" and "all sections of this paper." However, Natural Languages aren't as chaotic as it seems. Things tend to fall into a small number of classes. They can be structures, actions or time (many kinds of all of these).

The challenge in taking Natural Languages as the basis to producing a programming language is to decide which things should be referenceable in the context of computer programming, given the wide range of application domains. We should keep in mind that a naturalistic language should have an important property of most modern programming languages: it should be possible to construct abstractions on top of a relatively small number of primitive abstractions. Ideally, each application domain would build its own terminology and idioms, similar to what happens with Java APIs and similar to Natural Languages' dictionaries. What we propose here is that such primitive abstractions should be inferred from wider ground of Linguistics, rather than from computer engineering or mathematics or ad-hoc models such as objects. We propose this based on the fact that Natural Language comes before, and supports, all other domain-specific formal languages.

On a pragmatic vein, one fact has been clearly exposed by the wide adoption of Aspect-Oriented Programming: reflective and temporal references are important elements in programming. It is therefore logical to explore them even further. In our approach, we go back to the original, and more general, AOP idea described in [20]. For example, unlike AspectJ, statement-level anaphora, such as the one presented in the working example in Section 4.1.3, should be considered.

A more profound difference is that the emphasis in AOP was put on separation of aspects and components, and the reusability of aspects by different components. That design feature came from D and has proved to be very useful in practice, especially for development aspects such as tracing and profiling that are later removed from the final software product. However, because of that emphasis, the binding mechanisms in AspectJ don't use context information that could naturally be used. For example, expressions such as "the last operation" and "those bytes" "after reading [in a certain context]" should be supported. The emphasis should be the exploration of a variety of structural and temporal anaphora, some of which are captured by AspectJ, but most of which are not.

The anaphoric relations targeted here include not only intra-module referencing but also inter-module referencing. This may challenge the principle of modular programming. But, similar to what happens in AOP, if breaking the principle proves to be useful, then it means that the principle itself needs to be reformulated.

### 4.4 What This "Language" Is Not

The languages we're advocating are not for "end-user programming" (see related work section 5.4). While we believe that programs written in a naturalistic language will be more readable to non-programmers, our goal is not primarily to enable non-programmers to write computer programs. Nor is it for "natural language programming," an idea that has been around for some decades and that has been instantiated occasionally (e.g., [55], [2], [42], [60], [53]). We don't advocate implementing English! The languages we are proposing are naturalistic, but not natural. However, they will take their direction from the structure and expressiveness of natural languages rather than from the idealized models of traditional programming languages.

## 5. RELATED WORK

The ideas presented here have their roots in Aspect-Oriented Programming and the lessons we've learned from it. However, there are several fields of research, some of them considerably more mature than AOP, to which we must pay special attention.

### 5.1 Anaphoric Relations and Binding Theory

Researchers in computational linguistics and natural language processing have developed a sophisticated array of approaches to some of the problems that we are addressing, in the forms in which they occur in natural language. Anaphorical reference within natural language is the domain of binding theory, which draws its roots from Chomsky's pioneering work [8]. The problem that binding theory addresses is how to relate anaphoric expressions to their references; binding principles describe the relative positions of anaphors and their admissible antecedents in grammatical structure [4]. Chomsky's work proceeds from the observation that the two primary forms of anaphora (pronouns and anaphors, which are more complex referential expressions) correspond to forms (WH-movement and NP-movement) of syntactic movement. Alternative approaches to formal grammar, such as Head-driven Phrase Structure Grammar (HPSG), Lexical-Functional Grammar (LFG), or Categorial Unification Grammar (CUG), also must incorporate alternative, non-transformational (and less purely syntactic) accounts of anaphora (e.g., [51], [52], [7]).

While this work is clearly relevant, dealing as it does with the processing of richly expressive referential phrases of the sort that we would like to exploit, it's critical to recognize the difference between the analysis of naturally-occurring language, such as NLP must address, and the processing of restricted, formal, and artificial languages of the sort that we aim to develop. While there is much to learn from the natural handling of anaphoric reference, the language that we seek to develop is naturalistic but not natural. Therefore, our challenge is not to account for anaphora, but to exploit it, which reduces the challenge considerably.

### 5.2 Temporal Logic Programming

Most programming models and languages lack mechanisms for temporal reference. The notable exception is the work within the community of logic programming and the language generally associated with it, Prolog. Temporal logic programming has been proposed to reason about hardware and software systems (e.g., [50]). It has been used in the specification (e.g., [18], [29]), verification (e.g., [40], [46]), and synthesis (e.g., [12], [40]) of concurrent systems, as well as in the synthesis of robot plans (e.g., [14]). For a survey on temporal and modal logic programming languages, the reader may refer to [45].

While this work is relevant, its purpose is quite different from that of the work proposed here. Logic programming, in general, and temporal logic programming, in particular, focus on writing programs upon which certain theorems can be proved. While there are some lessons to be learned from formal specifications of time-dependent symbols in temporal logic, the language we seek to develop doesn't attempt at being used for proving theorems about the programs.

### 5.3 Cognitive Foundations of Programming Languages

The question of the degree of expressiveness afforded by programming languages, and the effectiveness of the notations in which programs are expressed, has been a topic of research investigation for some time. For example, studies in the psychology of programming have explored a range of issues including expert/novice differences in programming strategies [59], mental imagery used by programmers in thinking about programs [49], and the relationship of cognitive strategies to language features [58].

The use of intelligent systems to support learning programming languages has been the focus of a major research effort in the AI in Education community. In particular, a significant body of research, particularly arising in the UK, has investigated students' understandings of Prolog programs ([3], [5], [6], [11]). Prolog is a particularly interesting language to study, for a variety of reasons. First, for programmers used to procedural or functional styles, the declarative model that Prolog embodies can be a major challenge. Second, Prolog, being based on a logical calculus, has a superficial naturalism that can make it initially accessible to novice programmers. Third, for those novice programmers, Prolog rapidly becomes much more complex as the semantics of more advanced features such as "cuts" requires them to reconceptualize Prolog programs in terms of the sequential organization of search rather than in terms of a purely declarative formalism. These studies highlight the mutual influence of programming language structure and conceptual understandings on the part of its users.

One of the most influential analyses of the usability of programming languages is Greene's "cognitive dimensions" framework for notations ([15], [17]). The cognitive dimensions highlight the properties of notations in terms of the cognitive activities that they support, and so illuminate the questions of how and why notations "work" for particular sorts of tasks. For example, the dimension of viscosity [16] refers to a notation's resistance to change, and more generally, the complexity of making a single revision. A simple illustration of viscosity might be the insertion of a clause, such as an if or a while, around a block of code in a language that uses indentation to express structure (as in Python or Occam.) In these languages, encapsulating code inside a particular block involves changing the indentation of each newly-enclosed line of code. The notational device of using indentation to indicate block structure, then, has greater viscosity than the more conventional practice of indicating structure by using brackets. (However, the bracket approach may reduce visibility – the at-a-glance readability of the notation.) Greene and his collaborators have identified a range of relevant cognitive dimensions of notations, including premature commitment and role-expressiveness. The critical element of Greene's analysis is that it seems programs not simply as specifications of computer behavior, but as artifacts that people have to manipulate. It highlights the relationship between the program and the act of programming. Likewise, our approach is more concerned with expressiveness for the programmer rather than expressiveness for the computer.

In an effort to develop programming representations that bridge "the expressiveness gap," Pane ([47], [48]) studied the natural language descriptions of programming language tasks given by non-programmers. Pane was particularly interested in children's use of programming languages, although his methods and perhaps some of his findings apply more broadly. In an experimental setting, he had people give descriptions of programmatic behavior (in particular, the program for a Pacman-like game) and analyzed the forms of description that people produced. His findings pointed to a range of linguistic expressions by which people would describe the program's behavior, but which are poorly supported in conventional programming languages. For example, where people often produce complex grouping statements (such as "all the red objects" or "the objects on this side of the screen"), programming languages tend not to offer facilities for such dynamic groups, requiring iterative testing instead. Pane then went on to develop a programming language incorporating some of these elements.

In addition to the empirical approach that characterizes Pane's work, we feel that a theoretical grounding will be important for successfully developing this research. One promising and intriguing approach that we are beginning to explore in some preliminary work is the cognitive semantics perspective developed by Lakoff and others ([24], [25], [26]). Lakoff is a linguist and cognitive scientist whose work for many years has focused on the relationship between linguistic practice and cognitive capabilities. In particular, his studies of categorization (how people define and use classifications and categories) and of metaphor have begun to uncover a new way of understanding cognition. The central claim of cognitive semantics is that metaphor, rather than being a purely literary device, is in fact a central element of cognitive function. Metaphors typically occur not as individual elements of linguistic practice, but as entire systems of metaphors that relate different areas of experience. For example, the metaphor "LOVE IS A JOURNEY" reveals a complex structural mapping between domains, in which lovers are mapped to travelers, a relationship is mapped to a vehicle, shared goals are mapped to destinations, etc., and which accounts for a range of linguistic expressions such as "our relationship has hit a dead end," "I don't think we're going anywhere," "we're in high gear," "we were in the fast lane," "we hit a bump," "our relationship is on the rocks," etc. Through a series of detailed analyses, researchers in cognitive semantics have detailed the ways in which cognition is built upon a system of structural mappings between domains, of which these expressions are symptomatic. This applies not only to everyday cognition, but to more complex and abstract domains of reasoning which they demonstrate to be based through this metaphorical relation to embodied physical experience. Domains of application have included mathematics [28] and philosophy [27]. Our In some preliminary work, we are is beginning to explore the metaphorical structure of computer science, by analyzing the language used to describe and express computational concepts. This research suggests that the metaphorical model of cognition plays a strong role in Computer Science just as it does in other areas of reasining; metaphors of embodied experience such as "ITERATION IS MOVEMENT" and "DATA STRUCTURES ARE CONTAINERS" provide the foundation on which cognitive understanding of computation is based. We anticipate that these understandings will support the development of a language that is appropriately matched to everyday cognition.

## 5.4 End-User Programming

Although it is not the focus of our work, end-user programming is a related area of research because of its concern with forms of expression. End-user programming is inspired by the dual observations that, first, most software systems must be adapted by users, to some extent, to fit into their actual work; and, second, that although most people do not engage in programming in traditional languages, they certainly are adept at using many formal schemes. Nardi [43] discusses the use of such formal representations as knitting patterns and baseball scoring systems, and argues that there may be alternative formalisms which, suitably embedded in practice, will allow end-users to customize, program and adapt software systems; she cites the example of spreadsheet programming as an example [44]. Lave [30] has similarly observed that people who have difficulty with, say, mathematics in learning situations nonetheless can perform complex calculations in domains of everyday experience such as comparison shopping, currency exchange or calculating gambling odds.

One formalism that has been explored, especially in the area of programming environments for children, is graphical rewrite rules. KidSim [10] (subsequently called Cocoa and marketed as Stagecast Creator) and AgentSheets [54] are both systems for building interactive simulations based on graphical rule systems, and both have been successful, albeit in limited areas. Others have explored the use of Programming By Demonstration as a means to specify the behavior of software systems ([9], [33]). Programming by demonstration allows users to specify software systems through concrete operations rather than abstract description; however, the twin difficulties of generating appropriate generalizations and of conveying potential future activity to users have largely resulted in systems that are tightly coupled to specific domains, which have limited the uptake of the approach.

These approaches suggest that, despite decades of research into programming language design, there is still a great deal to learn not just about languages, but about programming, and especially about the relationship between the two. While those concerned with domain-specific languages or end-user programming have attempted to understand this relationship in order to make programming available to new communities of users, we believe that they are equally applicable to traditional programming practice.

## 6. CONCLUSION

The main goal of this paper was to re-generate some discussion around the role of Natural Languages in Programming Language design, and we tried to give a solid frame for this discussion. We believe this is an important topic for the problem of program understanding. The "end units" of any program are not only the microprocessors but also the human programmers. As such, it is only logical to take a serious look at the main form of human communication, namely Natural Languages. The power of Natural Languages is not so much the syntax but the way they allow us to organize ideas in "natural" ways. It is so much so that Natural Languages are, in fact, the primitive support for all other formal languages such as mathematical formalisms or microprocessor instructions. In other words, everything that can be expressed in those formal languages can be expressed in English, and not the other way around. This expressive power of Natural Languages is, to a great extent, supported by their sophisticated referencing and binding mechanisms, and those are precisely the focus of this paper.

We gave an informal example of a naturalistic programming language and analyzed some of its properties. At this point, this programming language is rather fuzzy, and many of the details will need to be worked out.

Further work includes a careful look at Linguistics and the existing models of Natural Languages. We will be looking for a variety of anaphora such as (1) pronouns, e.g. this, that, it, those, etc.; (2) object referents, e.g. the input stream, non-empty streams, etc.; (3) temporal referents, e.g. last, first, after reading, before encoding, etc.; (4) group referents, e.g. all, any; and (5) reflective referents, e.g. iteration, loop, operation, etc. We hope this study will give a solid framework for identifying primitive language mechanisms upon which we can design powerful programming languages that support not only a variety of programming models but also, and more importantly, natural program organizations within those models.

## 7. REFERENCES


[1] AspectJ Web site. http://aspectj.org

[2] Ballard, B. and Biemann, A. 1979. *Programming in Natural Language: NLC as a Prototype*. Proc. ACM/CSC-ER Annual Conference, 228-237.

[3] Bergantz, D., and Hassell, J. 1991. Information relationships in PROLOG programs: how do programmers comprehend functionality? Intl. Jnl. Man-Machine Studies, 35 (3), 313-328.

[4] Branco, A. 2001 Without an Index: *A Lexicalist Account of Binding Theory*. Proc. 8th Intl. HPSG Conference (Norway).

[5] Brna, P., Pain, H., and du Boulay, B. 1991. *Teaching and Learning Prolog: Supporting the Programmer*. Instructional Science, 20(2-3), 81-87.

[6] Brna, P., du Boulay, B. and Pain, H. 1999. *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*. Cognitive Science & Technology. Ablex.

[7] Chierchia, G. 1988. *Aspects of a categorial theory of binding*. In Oehrle, R., Bach, E. and Wheeler, D. (Eds), Categorial Grammars and Natural Language Structures, D. Reidel, Dordrecht. 125--151.

[8] Chomsky, N. 1973. *Conditions on Transformations*. In Anderson, S. and Kiparsky, P. (Eds), A Festschrift for Morriss Halle, 232-286. New York: Holt, Reinhart and Winston.

[9] Cypher, A. (ed.) 1993. *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press.

[10] Cypher, A. and Smith, D. 1995. *KidSim: End User programming of Simulations*. Proc. ACM Conf. Human Factors in Computing Systems CHI'95 (Denver, CO). New York: ACM.

[11] Duncan, D., Brna, P. and Morss, L. 1994. *A Bayesian Approach to Diagnosing Problems with Prolog Control Flow*. In Proceedings of the 4th International Conference on User Modeling, (Cape Cod, MA).

[12] Emerson E. A., and Clark E. M. 1982. *Using branching time temporal logic to synthesize synchronization skeletons*. Sci. Comput. Prog. 2.

[13] Ernst, E. and Lorenz, D.H. *Aspects and Polymorphism in AspectJ*. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pages 150-157,

Boston, Massachusetts, March 17-21, 2003. AOSD 2003, ACM.

[14] Georgeff, M. 1983. *Communication and interaction in multi-agent planning*. In Proceedings of the American Association for Artificial Intelligence. AAAI, Menlo Park, Calif.

[15] Greene, T. 1989. Cognitive Dimensions of Notations. In People and Computers V: Proceedings of HCI'89 (ed. Sutcliffe and Macaulay), 443-460. Cambridge: Cambridge University Press.

[16] Greene, T. 1990. *The Cognitive Dimension of Viscosity: A Sticky Problem for HCI*. Proc. IFIP Conf. On Computer-Human Interaction Interact'90.

[17] Greene, T. and Petre, M. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J. Visual Languages and Computing, 7, 131-174.

[18] Halpern, B., and Owicki, S. 1983. *Modular verification of computer communication protocols.* IEEE Trans. Communications COM-31.

[19] Kiczales, G., des Rivères J., Bobrow D. 1991. *The Art of the Metaobject Protocol*. MIT Press.

[20] Kiczales, G., Lamping, J., Mendhekar, M., Maeda, C., Lopes, C., Loingtier, J-M and Irwin, J. *Aspect-Oriented Programming*. In proc. European Conference on Object-Oriented Programming (ECOOP'97). Springer-Verlag LNCS n.1241. 1997.

[21] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. 2001. *An overview of AspectJ*. In proc. European Conference on Object-Oriented Programming (ECOOP'01). Springer-Verlag LNCS n.2072.

[22] Kojarski, S., Lieberherr, K., Lorenz, D.H., and Hirschfeld, R. *Aspectual Reflection*. In Proceedings of the AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, Massachusetts, AOSD 2003, March 17-21, 2003.

[23] Kojarski, S. and Lorenz, D.H. *Unplugging Components using Aspects*. In Proceedings of the ECOOP 2003 Eighth International Workshop on Component-Oriented Programming. Darmstadt, Germany, July 21, 2003.

[24] Lakoff, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. Chicago. University of Chicago Press.

[25] Lakoff, G. 1993. *The Contemporary Theory of Metaphor*. In Ortony, A. (ed), Metaphor and Thought (2nd ed). New York: Cambridge University Press.

[26] Lakoff, G. and Johnson, M. 1980. *Metaphors We Live By*. Chicago: University of Chicago Press.

[27] Lakoff, G., and Johnson, M. 1999. *Philosophy in the Flesh: The Embodied Mind and Its Challenge to Western Thought*. New York: Basic Books.

[28] Lakoff, G. and Nunez, R. 2000. *Where Mathematics Comes From: How the Embodied Mind Brings Mathematics Into Being*. New York: Basic Books.

[29] Lamport L. 1983. *Specifying concurrent program modules*. ACM Trans. Programming Languages and Systems 5, 2.

[30] Lave, J. 1988. *Cognition in Practice.* Cambridge: Cambridge University Press.

[31] Lieberherr, K., Lorenz, D.H., and Ovlinger, J. *Aspectual Collaborations: Combining Modules and Aspects*. The Computer Journal 46(5):542-565, September 2003.

[32] Lieberherr, K., Silva-Lepe, I., Xiao, C. *Adaptive Object-Oriented Programming Using Graph-Based Customization*. Communications of the ACM 37(5): 94-101. 1994.

[33] Lieberman, H. (ed.) 2001. *Your Wish is My Command: Programming by Example*. San Francisco, CA: Morgan Kaufmann.

[34] Lopes, C. and Lieberherr, K. 1994. *Abstracting Function-to-Process Relations in Concurrent Object-Oriented Applications*. In proc. European Conference on Object-Oriented Programming (ECOOP'94). Springer-Verlag LNCS 821.

[35] Lopes, C., 1996. *Adaptive Parameter Passing*. In Proc. International Symposium on Object Technologies for Advanced Software (ISOTAS'96). Springer-Verlag LNCS n.1049. Japan, 1996.

[36] Lopes, C. 1998. *D: A Language Framework for Distributed Programming*. Ph.D. Thesis, College of Computer Science, Northeastern University.

[37] Lopes, C. and Kiczales, G. 1998. *Recent Developments in AspectJ*. In proc. of Aspect-Oriented Programming workshop at ECOOP'98. Springer-Verlag LNCS n. 1543.

[38] Lopes, C. and Aguiar, P. 2003. *Acoustic Modems for Ubiquitous Computing*. In IEEE Pervasive Computing, Mobile and Ubiquitous Systems. Volume 2, Number 3, July—September.

[39] Lorenz, D.H. and Vlissides, J. *Pluggable Reflection: Decoupling Meta-Interface and Implementation*. In Proceedings of the 25th International Conference on Software Engineering, pages 3--13, Portland, Oregon May 3-10, 2003. ICSE 2003, IEEE Computer Society.

[40] Manna Z., and Wolper P. 1984. *Synthesis of communicating processes from temporal logic specifications*. ACM Trans. Programming Languages and Systems 6, 1.

[41] Manna Z., and Pnueli A. 1984. *Adequate proof principles for invariance and liveness properties of concurrent programs*. Sci. Comput. Prog. 4, 3.

[42] Miller, L.A. 1981. *Natural Language Programming: Styles, Strategies, and Contrasts*. IBM Systems Journal, 20(2), 184-215.

[43] Nardi, B. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: MIT Press.

[44] Nardi, B. and Miller, J. 1991. *Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development*. Intl. Jnl. Man-Machine Studies, 34, 161-184.

[45] Orgun, M. A. and Ma, W. 1994 *An overview of temporal and modal logic programming*. In Temporal Logic. First International Conference (ICTL'94), Lecture Notes in Computer Science No 827, (GABBAY D M and OHLBACH H J, Eds.), pp.445479, Springer Verlag, Bohn.

[46] Owicki, S. and Lamport L. 1982. *Proving liveness properties of concurrent programs*. ACM Trans. Programming Languages and Systems 4, 3.

[47] Pane, J., Ratanamahatana, C., and Myers, B. 2001. *Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems*. Intl. Jnl. Human-Computer Systems, 54, 237-264.

[48] Pane, J., Myers, B., and Miller, B. 2002. *Using HCI Techniques to Design a More Usable Programming System*. Proc. IEEE Symp. Human-Centered Computing Languages and Environments (Washington, D.C.), 198-206. Los Alamitos, CA: IEEE Computer Society.

[49] Petre, M. and Blackwell, A. 1999. *Mental Imagery in Program Design and Visual Programming*. Intl. Jnl. Human-Computer Studies, 51(1), 7-30.

[50] Pnueli, A. 1981. *The temporal semantics of concurrent programs*. Theoretical. Computer. Science 13.

[51] Pollard, C. and Sag, I. 1992. *Anaphors in English and the Scope of the Binding Theory*. Linguistic Inquiry, 23, 261-305.

[52] Pollard, C. and Sag, I. 1994. *Head-Driven Phase Structure Grammar*. Chicago: University of Chicago Press.

[53] Price, D., Riloff E., Zachary J. and Harvey B. 2000. *NaturalJava: A Natural Language Interface for Programming in Java*. Proc. ACM Intelligent User Interfaces Conference.

[54] Repenning, A. and Sumner, T. 1995. *AgentSheets: A Medium for Creating Domain-Oriented Visual Languages*. IEEE Computer, 28(3), 17-25.

[55] Sammet, J. 1966. *The Use of English as a Programming Language*. Comm. ACM, 9(3), 228-230.

[56] Shonle, M., Lieberherr, K. and Shah, A. 2003. XAspects: *An Extensible System for Domain Specific Aspect Languages*. In Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03) Domain-Driven Development track.

[57] Smith, B. 1982. *Reflection and Semantics in a Procedural Language*. LCS Technical Report. MIT.

[58] Soloway, E., Bonar, J., and Ehrlich, K. 1989. *Cognitive Strategies and Looping Constructs: An Empirical Study.* In Soloway and Iyengar (eds), Empirical Studies of Programmers, 23-251. Washington, DC: Ablex.

[59] Weidenbeck, S. 1985. *Novice/Expert Differences in Programming Skills*. Intl. Jnl. Man-Machine Studies, 23(4), 383-390.

[60] Winkler, D., Kamins S. and DeVoto, J. 1994. *Hypertalk 2.2: The Book*. Random House.