In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 18 Pages 53 - 67

Cognitive Perspectives on the Role of Naming in Computer Programs

Ben Liblit[1], Andrew Begel[2], and Eve Sweetser[3]

[1] University of Wisconsin, Madison WI 53706, USA, <liblit@cs.wisc.edu>
[2] Microsoft Research, Redmond WA 98052, USA, <andrew.begel@microsoft.com>
[3] University of California, Berkeley CA 94720, USA, <sweetser@berkeley.edu>

Abstract. Programming a computer is a complex, cognitively rich process. This paper examines ways in which human cognition is reflected in the text of computer programs. We concentrate on *naming*: the assignment of identifying labels to programmatic constructs. Naming is arbitrary, yet programmers do not select names arbitrarily. Rather, programmers choose and use names in regular, systematic ways that reflect deep cognitive and linguistic influences. This, in turn, allows names to carry semantic cues that aid in program understanding and support the larger software development process.

1 Introduction

Programming languages are designed to be precise, mathematical, unambiguous, and interpretable by machines. Natural languages, on the other hand, evolved over long periods of time to be interpreted by humans using a rich suite of cognitive processes. It would be wrong, however, to dismiss programming languages as wholly *un*natural. Programs are written by humans and do not proceed directly from keyboard to compiler; throughout development, humans read and modify the code that they and others have produced. Code that cannot be understood by a human is of little value, and human programmers are well aware that the code artifacts they produce must be readable as well as runnable. Thus, human cognition is reflected in the text of computer programs. To the extent that source code is expressed textually, code reflects linguistic cognition especially strongly. The purpose of this paper is to examine the impact of human cognition, and especially human language, on the text that forms computer software.

Among the many facets of software construction, this paper closely examines *naming*: the act of assigning identifying labels to programmatic constructs. This intensive use of invented words is a major deviation from natural language dialog, in which participants share a large, mutually understood, but relatively fixed lexicon. To a compiler, the choice of names is devoid of semantic meaning. Compilers have no understanding of natural language, so "`blue`" and "`Sgu9Asd1M`" are equally opaque, arbitrary sequences of letters. Provided that the programmer uses consistent spelling and capitalization, any name is as good as any other. Yet precisely because names are arbitrary, programmers have great freedom to select names that promote code understanding. We will examine several ways in which this is done, relating standard programming practice to modern theories of human language and cognition.[4]

[4] For a much more in-depth treatise on identifiers, see "The New C Standard" [14].

Our scope is narrow. We focus primarily on just four languages: C, C++, C# and Java, justified by the fact that these four languages are currently the dominant tools for large-scale industrial software development; any insight we gain, then, will have broad potential applicability. We use a variety of methodologies, such as morphological analysis, discourse analysis, grammar-based deconstruction of names, and metaphor-based deconstruction of names to discover the rationale behind programmers' choices for identifiers.

The remainder of this paper is organized as follows. Section 2 describes factors influencing the selection of individual names, and the way in which each name encodes useful information about its programmatic role. In Section 3 we show that names exhibit regularities derived from the grammars of natural languages, allowing them to combine together to form larger pseudo-grammatical phrases that convey additional meaning about the code. Section 4 explores two fundamental programming metaphors that aid understanding of code without direct linguistic interpretation. In Section 5 we review the phenomenon of overloading, a somewhat controversial programming feature that relates to polysemy and metaphorical extension of word meaning. Lastly, Section 6 reviews our findings and suggests directions for future study.

2 "Meaningful" Names

One of the most basic guidelines for writing good code is to use "meaningful" names. Humans create code and humans read code, and the names that programmers select reflect human cognitive structure. Naming engenders a strong reaction in software practitioners who advocate a particular naming scheme to ensure programs are understandable both with and without documentation [1, 4, 6, 22]. Jones asserts that memorability, confusability, and usability are three cognitively-based metrics by which to judge choice of names in a program [14].

In this section, we examine ways in which programmers select names that embed cognitively salient information. This practice provides other human beings with a rich source of hints about the behavior and intent of code.

2.1 Morphological and Metaphorical Regularities

Programmatic representation and manipulation of information is rather abstract. One routinely speaks of the "virtual world" that exists "inside" a computer as distinguished form the "real" world. We know that humans make pervasive use of metaphor to structure their understanding of the world, especially (though not exclusively) those facets of the world that are removed from immediate, bodily experience [18]. Metaphors can be useful for presenting domain tasks and for reifying abstractions into concrete terms, making them more accessible to people trying to comprehend programs [3]. They are central to naming program entities in a piece of software [8] as many studies on program comprehension have verified [2, 25, 28]. Names can also be classified into groups, called concept keywords, and associated with programming metaphors used during the design of data abstractions [24].

As an illustrative example, consider the collection of functions listed in Figure 1. These functions are part of the GNOME project, a collection of software tools for building desktop applications in the UNIX environment [9]. These functions are all written in C. For brevity, we show only the names of the functions, and elide their parameters, result types, and implementations. This short list of function names reveals certain regularities in their morphological structure and their use of metaphors.

```
gnome_druid_get_type          gnome_druid_set_buttons_sensitive
gnome_druid_new               gnome_druid_set_show_finish
gnome_druid_append_page       gnome_druid_set_page
gnome_druid_prepend_page      gnome_druid_set_show_help
gnome_druid_insert_page
```

Fig. 1. GNOME Druid Functions

The most obvious patterns are morphological. Each name contains several words (morphemes) separated by underscores. Although programmers have great freedom to choose names, technical considerations make it useful to disallow spaces and most punctuation marks. Underscores are allowed, however, and are often used by C programmers to delimit multiple morphemes in a single name. A second lexical convention, also popular in C, is to name functions, fields, and variables using only lower case letters. Distinct lexical conventions govern other kinds of names. Morphemes concatenated together (with no delimiting underscores) are commonly used for C#, C++ and Java names. Capitalizing each morpheme in the name (called Camel Case) is used for C++, C# and Java type names (e.g. `GnomeDruidPage`, `GtkContainerClass`[5]), and additionally for C++ and C# method names. Concatenating the morphemes together with underscores and fully capitalizing each word is commonly found in C and C++ macro names (`GNOME_DRUID_CLASS`, `GNOME_IS_DRUID`). It is also useful to be aware that Java lexical conventions differ slightly from C: Java uses embedded capital letters in field and method names (`indexOf`) instead of the underscores one would expect in C (`index_of`). Type names remain distinctive by their use of an initial capital letter (`Vector`). C++ is lexically transitional: some programmers prefer underscores while others favor embedded capitals.

Leading or trailing underscores mark internal names not intended for general use, and can be combined with other lexical conventions: `_GnomeDruid` is an internal type name; `__GNOME_DRUID_H__` is an internal macro. The distinctions drawn by these conventions coincide with profound differences in how these names are used in code.

Observe that all nine names above begin with `gnome_druid`. This lexical convention has a deeper metaphorical basis. When distinct pieces of code are combined, problems can arise if different programmers used the same names for unrelated things. Modern language facilities such as packages, modules, or namespaces help to encapsulate code and avoid these problematic "name collisions." The C language has no such facili-

[5] "GTK+" is the name of another software system upon which GNOME is layered. Thus, `Gtk` acts here as a single word.

ties, so instead we see programmers creating encapsulated namespaces through naming conventions. The preponderance of names defined by the GNOME software begin with `gnome` (or `Gnome` for types, or `GNOME` for macros), in effect creating a metaphorical container. The functions in Figure 1 all apply to a specific GNOME facility: druids. Thus, their names begin with `gnome_druid` or `GnomeDruid` or `GNOME_DRUID`, creating a namespace within a namespace; a box within a box.

Nested containers are a pervasive programming metaphor, which we revisit in Section 4. A second metaphor is expressed by the four function names that end with "`page`." A GNOME druid is a set of dialog boxes that guide the user through a linear sequence of steps. (The name "druid" is a humorous play on "wizard," Microsoft's name for a similar sequence-of-steps interaction popularized by Microsoft Windows.) A page, here, corresponds to a single step in the sequence. A druid contains an ordered, linear collection of pages. We have provisions for adding a page at the start of the sequence (`prepend`), the end (`append`), somewhere in the middle (`insert`), and for selecting which page is currently visible (`set`). The a metaphorical model, then, is a sheaf of paper folded open to reveal one page. The programmer adds and removes druid pages in the target domain just as one would add or remove real pages in the source domain. It is instructive that the programmer chose to call these "pages" and not "steps." The druid does not encapsulate a sequence of steps; it encapsulates the *presentation* of a sequence of steps to the user. Visual display is critical, and a page metaphor captures that more acutely; the fact that druid pages are rectangular, often with black text on a white background, only further enforces the mapping.

The GNOME druid is one small example of a programmatic abstraction with metaphorical correspondences, but it is by no means unique. Douce presents a long list of other metaphors found in computer programs [8].

2.2 Conflicting Pressures Impacting Name Length

If names are informative, then longer names, with more embedded subwords, should be more informative. Yet there are practical limits to the lengths of names, just as there are practical limits to the lengths of words in natural languages. Compilers for early programming languages such as FORTRAN limited names to six to eight characters, starting a long tradition of abbreviation in naming. Longer names may be more informative, but they are also more cumbersome to type and to read. Furthermore, long names quickly run up against finite display width:

```
distance_between_abscissae = first_abscissa - second_abscissa;
distance_between_ordinates = first_ordinate - second_ordinate;
cartesian_distance = square_root(
      distance_between_abscissae * distance_between_abscissae
    + distance_between_ordinates * distance_between_ordinates
);
```

Terse and abbreviated names, not even containing complete words, can enhance readability at the expense of role expressiveness [12] and require domain knowledge to decipher:

```
dx = x1 - x2;
dy = y1 - y2;
dist = sqrt(dx * dx + dy * dy);
```

Linguists have long observed that the character length of English words varies in inverse proportion to their frequency in written text [23, 34]. There are good reasons to believe that this tendency may be magnified in source code. First, names are concatenations of words, and therefore grow longer more rapidly than isolated words. Second, as seen above, typographic limitations make long names unwieldy in larger expressions. Third, programmers can sometimes leverage existing mathematical conventions that attach significant meaning to small names: consider x, y, z as spatial coordinates, or i, j, k as indices, or n as a counted number of items. Fourth, overuse of abbreviations can lead to a preponderance of unique symbols programmers must decipher [17], which may or may not lead to inhibited understanding [26, 27, 31]. Lastly, programmatic names vary in their semantic roles and visibility to other code; the need to be informative may be lessened for names that have narrow scope, or that are used in restricted ways.

How, then, do identifier names appear in the wild? We have collected statistical data concerning the lengths of names in a large body of Java, C, and C++ code. Our Java code represents the complete source to the standard Java v1.3 class libraries: 572,842 lines of Java code containing 83,750 defined names [30]. More than half of these (48,332) declare local variables or formal parameters. Local names have the narrowest scope, as each name exists only within the body of a single method. These names average only 4.7 characters with 1.3 embedded subwords (measured by the number of capitalized subwords), suggesting that the combination of heavy use and narrow scope incline programmers to be terse. The second largest semantic category consists of 17,575 public methods. These names average 12.1 characters with 2.4 embedded subwords. Public methods are fewer in number and sparser in use, but are visible throughout the entire program; this may justify their being given longer, more informative names.

Similar trends appear in C code. We have measured name lengths in the source code for Gnumeric, an open source spreadsheet application [10]. This corpus is 116,820 lines long with 22,740 declared names. As before, the most common names are local variables (9,872) and formal parameters (8,352), each with an average length of 4.7 characters and 1.2 embedded subwords (measured by the number of underscore delimiters in each identifier). We find it noteworthy that these metrics so closely match those for the Java corpus, since these two bodies of code have different authors, different purposes, and are written in different languages. Names with broader scope are both longer and less common: the 2,283 functions with file scope average 18.9 characters and 3.3 words, and the 1,358 functions with global scope average 20.5 characters and 3.6 words. Earlier we observed that C programmers use common prefixes to create artificial namespaces; manual inspection of Gnumeric's longest function names suggests that this practice may account for the greater average length of C function names compared to Java methods.

Lastly, analyzed the source code of Windows 2003 Server, which is written in C and C++. This corpus is around 45 million lines of code and contains 7,137,095 names. The names are almost evenly divided between global (3,690,597) and local (3,446,498). Local names average 7.6 characters and 1.8 embedded subwords (measured by the number of underscores or inter-capitalized subwords). Global names average 17.2 charac-

ters and 4.9 subwords, a significant increase. Public functions (as opposed to types) are slightly smaller, at 15.8 characters and 3.8 subwords on average, but there are many fewer of these than type symbols (858,421). This shows that on average, most types in Windows 2003 Server have one more subword than functions.

While these statistics are interesting, we must not read too much into them. Standard deviations are quite high; accordingly, we refrain from reporting on the remaining semantic categories or drawing detailed conclusions pending a more nuanced statistical analysis. A thorough treatment correlating name lengths, frequencies, visibilities, and semantic roles is beyond the scope of our present study. However, our crude preliminary analysis does suggest that there may be strong underlying principles that warrant closer examination.

3 Grammatical Sensibility in Name Use

Programmer-defined names do not exist in isolation. They interact with language specific punctuation and keywords to build expressions, statements, and other composite constructs. However, as we examine larger code fragments, we find evidence of natural language grammar throughout program text. Caprile and Tonella were able to analyze function identifiers by breaking them into individually meaningful words, classifying them into seven lexical categories, and further describing them by a sixteen-production grammar [5].

In this section, we consider ways in which names can create larger pseudo-grammatical utterances that further leverage natural language understanding to aid in code comprehension.

3.1 Names as Phrase Fragments

For purposes of this paper, we accept as given a metaphorical assumption that DATA ARE THINGS.[6] This mapping is reflected in Figure 2, which lists the constituent fields that make up a GNOME dock. Of nine fields, eight are nouns or noun phrases. The programmer has even made appropriate use of English plurals for precisely those fields whose data are lists rather than single items. The one unusual field is `floating_items_allowed`, an indicative phrase with an omitted "are" verb. This field holds a single true/false value. A human reading this code can immediately conclude that the field's value is true when "floating items are allowed" is a truthful statement about the dock. Thus, we have a specialized TRUE/FALSE DATA ARE FACTUAL ASSERTIONS mapping, which can override the more generic DATA ARE THINGS. As a practical matter, one is usually interested in the present state of an object, so these phrases are usually in the present tense. One does occasionally encounter past or future tense, though, when there is reason to inquire about past or future states.

We now turn our attention to a Java class, `java.util.Vector`. This class has three fields (`capacityIncrement`, `elementCount` and `elementData`), none of which are true/false. DATA ARE THINGS correctly predicts that all should be given noun phrase

[6] Text in SMALL CAPS should be read as metaphors.

```
container       bottom_bands     floating_children
client_area     right_bands      client_rect
top_bands       left_bands       floating_items_allowed
```

Fig. 2. Constituent fields of a GNOME dock

names. The methods of `Vector`, though, are more complex. Their names use a suite of schemes that reflect differing frames of understanding about the behavior and purpose of methods:

- METHODS ARE ACTIONS that actively change the state of the program. Methods obeying this model either return no value at all, or else return a value that provides only incidental information about the effect of the actions. Names for such methods are verb phrases in the imperative mood: `add`, `addAll`, `addElement`, `clear`, `copyInto`, `ensureCapacity`, `insertElementAt`, `remove`, `removeAll`, `removeAllElements`, `removeElement`, `removeElementAt`, `removeRange`, `retainAll`, `set`, `setElementAt`, `setSize`, `trimToSize`.
- METHODS ARE MATHEMATICAL FUNCTIONS that passively compute a result but do not alter the state of the program. Methods obeying this model must return some useful piece of data of interest to the caller. Such a method is identified with the value it produces, and therefore its name obeys the data naming principles given earlier:
  - TRUE/FALSE DATA ARE FACTUAL ASSERTIONS, so true/false returning methods have verb phrase names in the indicative mood: `contains`, `containsAll`, `equals`, `isEmpty`.
  - DATA ARE THINGS, so methods returning values other than true/false have singular or plural noun phrase names: `capacity`, `clone`, `elementAt`, `elements`, `firstElement`, `hashCode`, `indexOf`, `lastElement`, `lastIndexOf`, `size`, `subList`.

These conventions correctly describe thirty nine out of forty two methods in `Vector`. The exceptions are `toArray`, `toString`, and `get`, whose actual behaviors would properly place them in the category of non-true/false mathematical functions. `toArray` and `toString` each return an object equivalent to the original `Vector` but converted into a new form; their names are consistent with a specialized Java naming scheme used exclusively for conversion methods.

The `get` method is the strongest anomaly, as its behavior is to return a value but its imperative name suggests action. Furthermore, `get` is functionally identical to `elementAt`, which does have the expected noun phrase name. Yet another naming convention is at play here. It is common to offer matched pairs of `set`/`get` methods to manipulate an object's attributes. When the object has several attributes, method names are extended to identify the attribute they manipulate, such as `setColor`/`getColor`, `setName`/`getName`, and so on. Symmetric naming is informative: it reveals that the attribute exists and that the paired methods relate to that attribute in very specific way. However,

it also violates the DATA ARE THINGS principle by assigning imperative names to value-providing `get` methods. Programmers sometimes resolve this conflict inconsistently: `Vector` has `set`/`get`, but `setSize`/`size` instead of `setSize`/`getSize`.

3.2 Valence Cues

To characterize `copyInto` as a verb phrase is a bit generous, as it ends with a dangling preposition. In natural language terms, this fragment has one open valence slot: one additional item, specifically a noun, must be provided to complete the phrase. Java is not a natural language, but the valence cue is valid nonetheless. The `copyInto` method must be called with one additional argument: the array into which the vector's components are to be copied. Several other methods, such as `indexOf` and `elementAt`, contain similar preposition-based hints. Open valence slots can stem from many parts of speech: "all" is an adjective that requires a noun to modify, and `removeAll` expects a corresponding argument; "contains" is a transitive verb, and the `contains` method expects a single parameter that corresponds to the direct object in English speech.

Valence cues are not universal, and neither are they always in exact one-to-one correspondence with method arity. One `add` method requires two parameters while another requires just one, yet they share the same name. The `subList` method might instead have been called `subListFromTo`, with two open prepositions to emphasize the two required arguments. Clearly there is room for variation, and the programmer must balance the benefit of valence cues against other concerns for length and readability. The design of the underlying programming language is significant as well. While the majority of programming languages treat method names as atomic, the object-oriented language Smalltalk is a noteworthy exception. Smalltalk method names contain multiple words delimited by colons, with one argument after each segment. A Smalltalk version of `insertElement` would likely be named "`insert:at:`", and would be used as:

```
roster insert: newHire at: position
```

Contrast with Java or C++ syntax:

```
roster.insertElement(newHire, position);
```

A Smalltalk method name is woven in among its arguments. This is not merely suggested; it is required. A Smalltalk programmer *must* produce some name fragment to appear before each and every argument. English words with open valence slots are an obvious choice. Hence one finds Smalltalk method names like "`inject:into:`" or "`moveTower:from:to:using:`" or the highly descriptive "`scheduleArrivalOf:-accordingTo:startingAt:`" [11, 15, 19].

In object oriented languages, data are personified as active agents with internal state (fields) and a set of exported behaviors (methods). To invoke a method, then, is to direct a particular agent to perform a particular action or perform a particular calculation with respect to itself. Methods with verb names have an open valence slot for a subject; this slot is filled by the object whose method is being called. In the "`roster.insertElement(...)`" example given above, `insertElement` reads as an

imperative command issued to `roster`. That the subject appears immediately before its verb helps to create extended pseudo-grammatical utterances for English speaking programmers, as in:

factual assertion: the roster contains the record	`roster.contains(record)`
factual assertion: the roster is empty	`roster.isEmpty()`
imperative command: roster, remove all junk	`roster.removeAll(junk);`
imperative command: roster, set your size to five	`roster.setSize(5);`

For noun-named methods, a reading in the possessive case comes equally easily, provided that the reader has access to the English "'s" construction:

computed attribute: the roster's first element	`roster.firstElement()`
computed attribute: the roster's capacity	`roster.capacity()`

A possessive reading becomes less natural for more complex expressions; we revisit this issue in greater depth in Section 4.

4 Containers and Paths

We now broaden our scope to issues of cognition beyond language comprehension. While it is not our intent to undertake a general review of cognition in computing, there is one pair of models that directly impact name use and require closer inspection: OBJECTS AS CONTAINERS and POINTERS AS PATHS.

Until now, we have been describing objects predominantly using a container metaphor, such as when we speak of objects' "constituent" fields, or "internal" state versus "exported" behaviors. This model is consistent with real-world metaphors that treat composite entities as containers that enclose their attributes. The metaphor carries over to visual depictions of data structures, which typically show objects as rectangles partitioned into smaller boxes, one for each member field. Objects within objects are depicted as boxes within boxes.

However, objects need not only enclose one another in a strict containment hierarchy. One object may instead reference another indirectly, using what C calls a *pointer* or what C# and Java call a *reference*.[7] A pointer uniquely identifies a single piece of data, but rather than holding the data itself, a pointer simply records where the data may be found. In pictorial representations, pointers are presented as arrows from referrer to referent. A C or C++ programmer must explicitly decide whether each part of a complex data structure embeds another directly or instead uses indirect reference by way of a pointer. Java programmers do not deal with this directly, as Java allows indirect reference only; embedding is not provided for compound data types.

In C and C++, a dot (`.`) marks access to an embedded field of an object. If `dock` is a GNOME dock, then "`dock.container`" accesses its embedded `container` field. When fields are themselves composite, field access expressions can be extended as long

[7] C++ offers both pointers and references. C++ references roughly combine C pointer semantics with Java reference syntax, and may be seen as a transitional form.

as necessary to access deeply embedded subfields, as in `dock.container.widget.requisition.width`. Section 3.2 suggested that the English possessive "'s" construction can capture member access. While that is satisfactory for simple expressions like `roster.firstElement()`, it is difficult to claim that "the dock's container's widget's requisition's width" is an effective way to comprehend extended field access chains. We believe that programmers do not consider such expressions in explicitly linguistic terms. Rather, the operative model is of drilling down from outer containers into inner ones: start at the dock; from there, go down into its container; from there, continue down into its widget; from the widget, ...

This hypothesis gains credence when we consider pointer fields. Instead of a dot, C and C++ represent pointer accesses using "`->`", a textual approximation of an arrow. The `client_area` field of a GNOME dock is a pointer, so in "`dock.client_area->parent`" we use "`->`" to cross that pointer and retrieve its `parent` subfield. Field access expressions are driving directions: they describe a path from a starting point to a final destination, with individual steps along the way listed in the order they will be traversed. Programmers casually refer to "following" or "crossing" a pointer, reinforcing a spatial metaphor in which pointers are paths or bridges between islands of data. One can even refer to "falling off the end" of a data structure by traversing one pointer too many.

4.1 Object Orientation, Anaphora, and the Role of Deixis

If we wish to use the `lastElement` method of a Java `Vector` named `roster`, we write `roster.lastElement()`. However, within the code that implements `lastElement`, it is no longer necessary to refer to `roster` by name. Within the body of `lastElement`, we can call `size()` or read `elementData` or access any other member fields and methods without explicitly naming the containing object. Instead, the containing object is implicitly assumed to be the same object that the `lastElement` method itself was called upon.

Since the methods of single object tend to be tightly interdependent, this convention is quite convenient and greatly improves code readability. It also creates an interesting deictic shift: we are at a different place in the metaphorical data space. We are, in some sense, "inside" the object. This new location gives us certain benefits and privileges. We can access internal attributes of the object that are not visible to outsiders. And we can use abbreviated names, such as `size()` instead of `roster.size()`, because the missing information is implicit in the surrounding context. Should we need to refer to the object we are inside, C++, C# and Java each supply a "`this`" keyword for that purpose. Smalltalk uses "`self`". The author of Gnumeric uses "`me`". (Gnumeric is written in C, but with a strongly object oriented style.) Java also allows referencing the object's superclass through the keyword "`super`." This is useful when the current class has hidden access to the superclass' fields or methods.

Within an object, the meaning of `this` is constant; it may not be altered through execution of a program statement that may introduce the name of a new object. By contrast, in natural languages, once the subject of a sentence has been introduced, subsequent sentences often anaphorically refer to the subject using a pronoun, such as

"she" or "it" (e.g. "Nancy brought some chips to the party. *She* also brought a bottle of soda."). Detienne notes that anaphora is rare in programming languages [7],[8] yet anaphoric references to temporal conditions (e.g. "before the execution of this method") can be found in aspect-oriented programming [20]. Incorrectly applied, anaphora can lead to ambiguity of reference, which is unacceptable in programming languages.

5 Polysemy, Homonymy, and Overloading

The last phenomenon we address is *overloading*: the policy of sharing one name among several functions. C does not permit overloading, but C++, C# and Java do. The Java `Vector` class uses overloading: it contains two `add` methods, two `addAll` methods, two `indexOf` methods, two `lastIndexOf` methods, two `remove` methods, and two `toArray` methods. In many cases, same-named methods accept different numbers of arguments. A few overloaded methods have identical arity but differ in the expected types of their arguments. These distinctions allow the compiler to select the appropriate method, and any overloading that cannot be disambiguated in this manner is detected and disallowed.

To see more aggressive use of overloading, we turn to KWord, an open source word processor written in C++ [16]. The programmer has defined a `KWString` type to represent text sequences within a document. This type contains several overloaded methods, including eight named `insert`. Every `insert` method expects two arguments. The first argument is always the position within the `KWString` where the insertion should take place. The second argument may be:

- a sequence of characters as represented by the underlying graphical toolkit
- a sequence of characters as represented by KWord
- a single character
- a picture
- a tab
- a placeholder for automatically generated text, such as the current date or page number
- a footnote
- an anchor for a floating figure

The programmer could have given each method a distinct name: `insertPicture`, `insertTab`, `insertFootnote`, and so on. The use of a single name creates a conceptual grouping. These methods are all, in some sense, equivalent. Each one inserts a document fragment at a given position; they differ only in what is being inserted. Using the same name for all methods is just as sensible as the same English verb in "insert a picture" and "insert an anchor". For English, we could argue that there is only a single "insert" verb, because "a picture" and "an anchor" are grammatically interchangeable. In the KWord program, pictures and anchors are quite distinct, and insertion requires slightly different code for each. But the general effect as seen by the caller is the same in all cases, making it convenient to simply call `insert` and let the compiler determine which method is intended. In effect, `insert` is polysemic.

[8] Exceptions are found in command-line shells, where it is possible to refer to the result of the last evaluated expression (e.g. `$?` in `csh`).

5.1 The Operator Overloading Debate

Operator overloading takes the view that mathematical operators are simply functions with special names and syntax: "`x + y`" is no different from "`sum(x, y)`". Certain operators, like addition of two integers, are built in to any sensible language. But there may be other forms of addition not anticipated by the language designers. Programmers may wish to define arithmetic on a data type, for example, a representation for Hamilton's quaternion algebra, where the intended behavior of "+" is obvious. We have a metaphorical extension of addition into a novel realm. In languages that support operator overloading, the programmer can extend the basic operators to support new data types; defining addition of quaternions is similar to defining any other function [29].

Operator overloading is controversial. Its supporters argue that overloaded operators improve readability: "`q + r * s`" is much easier to scan than "`sum(q, product(r, s))`" or the more object-centered "`q.plus(r.times(s))`". Authors of numerical and scientific code find operator overloading especially useful [33]. Skeptics counter that an overloaded operator can perform arbitrary computation having no relationship to its more basic form, and that in the presence of operator overloading, even the simplest code fragments cannot be assumed to have any consistent meaning: "`x + y`" could contain an entire word processor.

C has no overloading at all. C++ and C# offer overloading for names as well as for operators. Overloadable operators in C++ and C# include the obvious arithmetic ones (`+`, `-`, `*`, `/`) but several that are more obscure. We can define our own "`[]`" operators to create objects that act like arrays. Overloading "`->`" (in C++) lets us create so-called "smart pointers." Even function invocation, the "`()`" operator, can be overloaded to create data objects that may be called as though they were functions. Java allows method name overloading only; operator overloading is not available.

In cognitive terms, we can deconstruct this debate into two distinct concerns. First, there is the issue of the arbitrariness of overloaded operators' behavior; the worry that overloaded operators may diverge into homonyms with no rational polysemic connection to their primitive counterparts. While it is true that "`x + y`" could be hiding an entire word processor, it is equally true that `insert` could remove data or that `Vector` could represent the color blue. Skilled programmers use names in cognitively rational ways, and we have every reason to believe that they can do the same for operators. The second concern is that overloaded operators make it harder to infer the behavior of code. This is a more difficult matter, as programmers read code in different ways with different goals. When it is important to understand the precise, "literal" behavior that the code will create when run, overloaded operators can be obfuscating. When the intent is to understand code at a higher level of abstraction, operators that have been extended in a metaphorically principled manner can be a boon.

5.2 Homonymic Extension of the Shift Operators

Surprisingly, the most widespread use of C++ operator overloading is homonymic. C++ defines two standard operators that manipulate the bit-level representation of numbers: left shift (`<<`) and right shift (`>>`). The motivation for the built-in operators is clearly

based upon visual iconicity: << and >> resemble arrows pointing left and right, respectively. These are binary operators: "`x << 4`" shifts the bits representing `x` left by four positions.

The standard C++ library overloads the shift operators for a completely unrelated purpose: file input and output. One writes "`file << data`" to store data into a file, and "`file >> data`" to load data from a file. Programmers can add their own << and >> operators and thereby extend file I/O with their own data types. This is a technically clever trick, but there is nothing that particularly justifies output as a polysemic extension of shifting left, or input as a polysemic extension of shifting right. "`file << data`" simply looks like data flowing across an arrow into the file. The visual metaphor is compelling enough to justify what would otherwise be derided as "arbitrary" overloading.

6 Conclusions and Future Work

Modern software is incredibly complex. The source to Microsoft Windows XP has 40 million lines of code [21]. The Linux 2.6 kernel weighs in at 6 million lines [32]. Managing this kind of complexity requires that programmers draw deeply upon all their cognitive abilities. We have discussed several ways in which programmers select and use names in cognitively motivated ways. Lexical and morphological conventions convey basic information about a name's role, while metaphors encourage productive inferences drawn from other domains of experience. The grammars of natural languages shape name use in intricate ways, and polysemy appears as overloading with attendant debates over literal versus metaphorically extended meaning. Throughout, we find that programmers leverage fundamental aspects of cognition and natural language comprehension to make code easier to read and understand.

Practical considerations have motivated us to narrowly study the role of naming in imperative languages. If we were to expand our scope, we might ask how increasingly linguistically sophisticated programming languages have changed the cognitive burden on programmers. Conversely, we might ask how language designers could better support programming as a cognitive, communicative task. Should programming languages provide linguistic support for anthropomorphism, as indicated by Herbsleb's study of metaphors used to describe software behavior [13]?

Programming is a sophisticated intellectual process that combines aspects of natural language with the regular structure of formal mathematical thought. The study of programming truly has the potential to contribute valuable perspectives to current research in linguistics and cognition.

References

1. N. Anand. Clarify function! *SIGPLAN Not.*, 23(6):69–79, 1988.
2. M. Ben-Ari and J. Sajaniemi. Roles of variables as seen by CS educators. In *Proceedings of ITiCSE '04*, pages 52–56, New York, NY, 2004.
3. A. F. Blackwell. Metaphor or analogy: How should we see programming abstractions? In P. Vanneste, K. Bertels, B. D. Decker, and J.-M. Jaques, editors, *8th Annual Workshop of the PPIG*, pages 41–49, Ghent, Belgium, April 1996.

4. L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, H. Spencer, D. Keppel, and M. Brader. *Recommended C Style and Coding Standards*. <http://www.psgd.org/paul/docs/cstyle/cstyle.htm>.
5. B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of WCRE '99*, page 112, Washington, DC, 1999.
6. B. Carter. On choosing identifiers. *SIGPLAN Not.*, 17(5):54–59, 1982.
7. F. Detienne. *Software Design – Cognitive Aspects*. Springer, 2001.
8. C. Douce. Metaphors we program by. In *16th Annual Workshop of the Pyschology of Programming Interest Group*, Carlowe, Ireland, April 2004.
9. The GNOME Project. *GNOME libraries release 1.0.6*. <http://www.gnome.org/>.
10. The GNOME Project. *Gnumeric release 0.64*. <http://www.gnome.org/projects/gnumeric/>.
11. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Co., Reading, 1983.
12. T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of HCI '89*, Cognitive Ergonomics, pages 443–460, 1989.
13. J. D. Herbsleb. Metaphorical representation in collaborative software engineering. In *Proceedings of WACC '99*, New York, NY, USA, 1999. ACM Press.
14. D. M. Jones. *The New C Standard: An Economic and Cultural Commentary*. Derek M. Jones, 2003.
15. T. Kaehler and D. Patterson. *A Taste of Smalltalk*. W. W. Norton & Company, Inc., New York, 1986.
16. The KOffice Project. *KWord release 1.0*. <http://www.koffice.org/kword/>.
17. K. Laitinen. Estimating understandability of software documents. *SIGSOFT Softw. Eng. Notes*, 21(4):81–92, 1996.
18. G. Lakoff and M. Johnson. *Metaphors We Live By*. University of Chicago Press, Chicago, 1980.
19. S. Lewis. *The Art and Science of Smalltalk*. Hewlett-Packard Professional Books. Prentice Hall International Ltd., Hertfordshire, 1995.
20. C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr. Beyond AOP: toward naturalistic programming. In *Companion Proceedings of OOPSLA '03*, pages 198–207, New York, NY, 2003.
21. V. Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional, September 2005.
22. D. Marca. Some pascal style guidelines. *SIGPLAN Not.*, 16(4):70–80, 1981.
23. G. A. Miller and E. B. Newman. Tests of a statistical explanation of the rank-frequency relation for words in written English. *American Journal of Psychology*, 71:209–218, 1958.
24. M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *Proceedings of MSR '05*, New York, NY, 2005.
25. J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of HCC '02*, page 37, Washington, DC, 2002.
26. S. B. Sheppard, B. Curtis, P. Milliman, and T. Love. Modern coding practices and programmer performance. *IEEE Computer*, 12:41–49, Dec. 1979.
27. B. Shneiderman. *Software Psychology*. Winthrop Publishers, 1980.
28. E. K. Soloway. An empirical investigation of the tacit plan knowledge in programming. In J. C. Thomas and M. L. Schneider, editors, *Human Factors and Computer Systems*, pages 113–133. Ablex, Norwood, NJ, 1984.
29. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Co., Reading, third edition, 1997.
30. Sun Microsystems, Inc. *Java 2 SDK, Standard Edition, release 1.3*. <http://java.sun.com/j2se/1.3/>.

31. A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.
32. L. Torvalds et al. *Linux release 2.6.0*. `<http://www.kernel.org/>`.
33. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, Sept. 1998. Special Issue: Java for High-performance Network Computing.
34. G. K. Zipf. *Selected Studies of the Principle of Relative Frequency in Language*. Harvard University Press, Cambridge, 1932.