Metaphors we Program By: Space, Action and Society in Java

Alan F. Blackwell

University of Cambridge Computer Laboratory
William Gates Building, Cambridge CB3 0FD, UK
Alan.Blackwell@cl.cam.ac.uk

Abstract. A corpus analysis of the standard Java documentation revealed the range of conceptual metaphors shared by library authors and users of packages such as java.util and java.bean. These metaphors included the expected mental models of internal program behaviour, but also consistent references to a spatial image-world with material properties and flows. More surprisingly, program components are metaphorically understood as actors with beliefs and intentions, working together according to social relationships. Rather than mechanical imperative models or mathematical declarative ones, it seems that one of the most widespread bases for conceptual models of programming is of social entities that act as proxies for their developers. This may have significant implications for the design of new programming languages and environments.

1. Introduction

Most people who write computer programs have some kind of understanding of what will happen inside the computer when it runs their program. This *mental model* [16] is derived from textbooks, from conversations with other programmers, from commonsense interpretation of the language semantics, and from system documentation. Wherever it comes from, just as with the mental models that users have of other technical systems [5], one certainty is that the programmer's mental model of program behaviour will only partially resemble that of the expert computer scientists who originally designed the language. In order to improve the usability of programming systems, designers need to understand the nature of the mental model that users have of program behaviour, and hopefully use that knowledge to assist learning, improve user interface design, or even modify the language specification.

A great deal of previous research at PPIG has investigated mental models of programs, both among students who may become professional programmers in future, and among end-user programmers. These approaches have focused on the program execution model, and on the way that it is understood through the user interface of the programming environment. This paper has a similar concern, but addresses a different aspect of the programming environment, and a different means of communicating with the programmer. Rather than virtual machines, execution models or programming language semantics, this paper addresses the basis for mental models of standard component libraries. Expertise in a particular programming language

increasingly depends on understanding and application of such libraries, yet only a few studies [4,12] have focused on usability of libraries or APIs. So far as I am aware, this paper is the first systematic study that has investigated mental models of programming libraries. This topic is clearly relevant to improving the usability of programming environments. It also offers a new perspective on the "ecology" of the end-user programmers' mental models, because of the wide range of people who write libraries, including open-source developers, professional technical authors, and specialist engineers. The underlying concepts of a new programming language are relatively coherent, often having been developed by a single person. In contrast, the extensive libraries of a language such as Java have been written by a large community of developers, and represent a collective rather than individual design.

2. Conceptual metaphors as the basis for mental models

The research method applied in this study was originally developed in the field of cognitive anthropology, although it is now widely used in applied linguistics, education, psychology, philosophy and other fields. Popularised by George Lakoff and Mark Johnson in their book "Metaphors we Live By" [7], it analyses the structures and vocabulary of ordinary language in order to identify underlying *conceptual metaphor* schemata. A typical example of such analysis might observe that the phrase "my confidence is rising" does not refer literally to my climbing the stairs or riding a balloon, but is a metaphor employing the common schema that INCREASE IS UPWARD MOTION (in the conceptual metaphor literature, a schema is conventionally identified by typesetting in small caps). Indeed, almost all our everyday abstract language is found to rely on metaphors of physical motion, space or the body [6], although conceptual metaphor analysis also reveals schemata derived from social relations and other experience.

Many works have been published in the field of conceptual metaphor studies since Lakoff and Johnson's seminal book. A typical survey, including both case studies and advice on research methods, is that by Cameron & Low [3]. Among HCI researchers, the best-known applied example is probably an analysis of the desktop metaphor by Tim Rohrer, a professional philosopher specializing in conceptual metaphor [13]. A more substantial analysis of the relationship between conceptual metaphor theory and user-interface metaphor as promoted in HCI is forthcoming [2].

The usual method of conceptual metaphor research is to take some corpus of human discourse, either textual or an oral transcript, and code each utterance according to the "literal" meaning of the vocabulary. However the term "literal" must be treated with caution, because all language has evolved from many layers of dead metaphor [9,14]. The analysis therefore depends on some (perhaps implicit) hypothesis. For example, if the hypothesis is that all abstraction must be traced to a visuo-spatial image (the image-schematic view), then an analyst might not stop with a familiar abstraction, but could consider archaic derivations of the word that may be unknown to the modern user. The text of this paragraph itself contains such examples: "depend" originally means "to hang from", while "abstract" means "to pull away from". The justification for "mining" vocabulary to this degree seems problematic,

when claiming to represent mental models of the user, so I have avoided extreme or archaic interpretations in this study.

3. Method of this study

This study reports a conceptual metaphor analysis of the Java documentation. The original intention was to identify the extent of spatial imagery in the descriptions of computational abstraction in the Java libraries, motivated by [1]. The method used was influenced by this spatial imagery hypothesis, although as shall be seen, the main findings did not support that hypothesis.

The corpus used for analysis was the Java documentation (javadoc) that is distributed with the standard Sun Java SDK release. The version of the SDK used was the Java 2 platform, standard edition 5.0 [15]. The full documentation in this release comprises more than 64MB of HTML text, so it was necessary to select a subset for this initial study. Motivated by the spatial imagery hypothesis, I wished to focus on the fundamental computational models of everyday Java programming, and to avoid functions that one might already expect to be based on visual or spatial images. For example, I did not code the AWT (abstract windowing toolkit) package, as it would be expected to describe images and spatial layout of the screen. I chose to focus on three packages that express the Java computational model, and would be highly familiar to Java programmers: the applet package, the beans package, and the util package.

The java.applet package is a simple one, including only one class containing 25 methods, plus 3 other interfaces. It is to some degree obsolete, as most programmers would now use the Swing equivalent JApplet, but is useful in providing a fairly pure description of basic architectural concepts in Java, independent of graphics management considerations. The total documentation in java.applet is about 800 lines. I used this as a pilot corpus, in order to develop the coding procedure used for the other two packages. The java.beans package is somewhat larger, including 26 classes, 9 interfaces and 2 exception types. This provides a more contemporary view of Java system architecture, used by many programmers every day. The documentation of java.beans totals about 5500 lines. I coded this corpus by hand, in order to ensure that all terminology was seen and interpreted in the context where it was used. Finally, the java.util package is one of the largest, and most frequently used, in the Java distribution. It includes 49 classes, 16 interfaces and 20 exception types. The documentation totals around 20,000 lines. I analysed this larger corpus automatically, using techniques that were informed by the manual analysis of the two smaller corpora.

3.1 Coding technique

The main objective in each analysis phase was to inspect the characteristic vocabulary used by the javadoc authors, in order to identify candidates for conceptual metaphors. This involved collating every word in every sentence of the documentation, apart

from conjunctions, tense markers and other grammatical elements of English. In the pilot java.applet corpus, the resulting vocabulary was surprisingly small – only 90 unique words, after elimination of alternate conjugations of each word, and of some other word categories that are described in the results section as being highly conventionalised.

I wished to focus on natural language descriptions of program behaviour, so did not analyse identifier names. The recent study of identifier names by Liblit, Begel and Sweetser [8] is therefore complementary to this study. I also omitted descriptions of application domains, or any aspect of the program involving interface to the external world. I coded every remaining sentence, including the introductory description of the package and each class, descriptions of methods, variables, interfaces, exceptions and parameters. All of these appeared in the documentation according to javadoc conventions, including some heading terms that are repeated continuously (class, method and so on).

Javadoc is generated automatically from comments in the program source code, so it can be assumed to reveal the mental model of the programmers who implemented the Java libraries and wrote original comments. Of course, as the main documentation for the Java product release, we can also assume that it has been edited by professional editors in order to improve its consistency and usability by ordinary Java programmers. This is evident in, for example, the repetition of conventional phrases when the same parameter must be documented in the context of a dozen different methods. When coding manually, I coded only a single occurrence of each repeated passage. It was not so easy, in the automated third phase of analysis, to identify and ignore such repeated texts. However the volume of text considered in the third phase is sufficient that local repetition has not had any significant effect on relative word frequency.

In the pilot phase, I had been expecting that it would be necessary to read past many occurrences of Java language keywords. In fact, these seldom appeared. Instead, it became apparent that the idiom of javadoc writing is heavily influenced by the conventional vocabulary of computer science textbooks, especially those describing object-oriented software design. In the second and third analysis phases I considered this jargon separately from other vocabulary, firstly because it is so frequent, and secondly because the conventionalised nature of the idiom means that it may not be fully interpreted either by writers or readers. Nevertheless, conceptual metaphor analysis does take an interest in such conventionalised idiom, so although it is less surprising, because more obviously synthetic, I summarise that content also.

The manual coding of the java.beans package resulted in a collection of about 330 unique terms. These were obtained by reading all 5,500 lines of documentation one sentence at a time, identifying every word that was not a grammatical element or part of the conventional Java jargon already isolated in the pilot phase, and counting occasions on which that term was repeated later in the package. As explained, the words in duplicated phrases and sentences were only counted when they first appeared. Common duplications included the documentation for many slightly varying constructors in a single class, or certain standard parameters that were repeatedly documented in the context of many different methods.

The automated coding of java.util in the third phase provided a larger corpus of data that could be used to verify the results of the second phase. However automated

coding brought disadvantages, in that it was not possible to make a decision on each word in the context in which it appeared. Instead, all words in the documentation files were uniquely identified and counted, resulting in a list of about 4,000 unique words extracted from the 20,000 line corpus. Basic grammatical elements were eliminated from this list, as were all compound words (assumed to be identifiers), and references to application domain entities such as geographic locations, calendar and time terminology, mathematical terms, financial terms and so on. The remaining words were sorted and collapsed to combine different prefixes, postfixes and other conjugations of a common root word. The final list used for analysis comprised 1,100 unique terms, including some categories that were not explicitly counted in the first two phases (for example, the word "class" appears 899 times in the documentation of java.util, and the word "element" appears 1858 times). The corpus will have included some homonyms (e.g. the verb "turn" for rotation, and the noun "turn" for alternation), and the automated analysis in phase 3 did not distinguish the proportion of times each meaning has been used. My coding was simply based on the first definition appearing in a standard dictionary, so long as this was consistent with the computational domain. Judgments were not checked by dual coding, although some problematic cases were discussed with colleagues. As discussed later, this may weaken the results, although consistent with much common practice in conceptual metaphor research.

4. Results

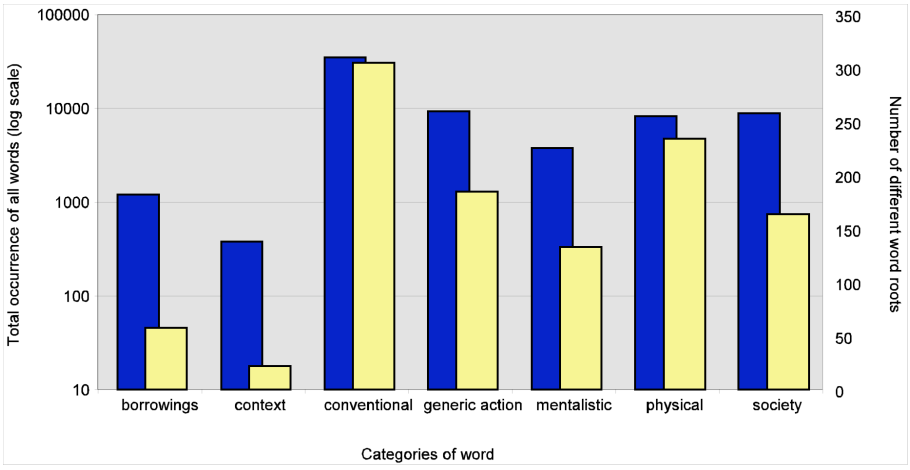

Fig. 1. **Overall coding, showing total occurrences (dark blue bars, left Y axis) and individual words (light yellow bars, right Y axis) in each category**

The results are discussed using several broad categories of word classification. These categories are shown in Figure 1, which indicates the relative size of each category as

found in the largest corpus (java.util). In this figure, and all those that follow, the category size is compared according to two different statistics. The first, shown on the left axis, shows the total number of times that any word in that category appears in the corpus. The most common words appear very frequently indeed (one would expect some approximation to the Zipf distribution), so all comparisons of frequency are made on a logarithmic scale. The left hand axis is therefore calibrated on a log 10 scale. The second statistic is the number of different word roots that are included in that category. In figure 1 we can see, for example, that although there are a small number of words in the "context" category, these appear very frequently in the corpus. These two statistics can be interpreted as giving an indication of the predominance (occurrences) and richness (different words) of the respective conceptual metaphors. A reviewer of this paper has noted that similar measures, known as "type-token ratios" are studied in statistical linguistics.

The seven overall coding categories shown in Figure 1 are broken down further in the next seven sections of the paper (unfortunately in a different order to which they appear on the X axis of Figure 1). In this discussion, a number of conceptual metaphors are presented. The narrative description is based on my qualitative reading from the first and second phases of analysis. The quantitative indications of predominance and richness within each metaphor are based on the third phase.

4.1 Conventional Terminology of Programming

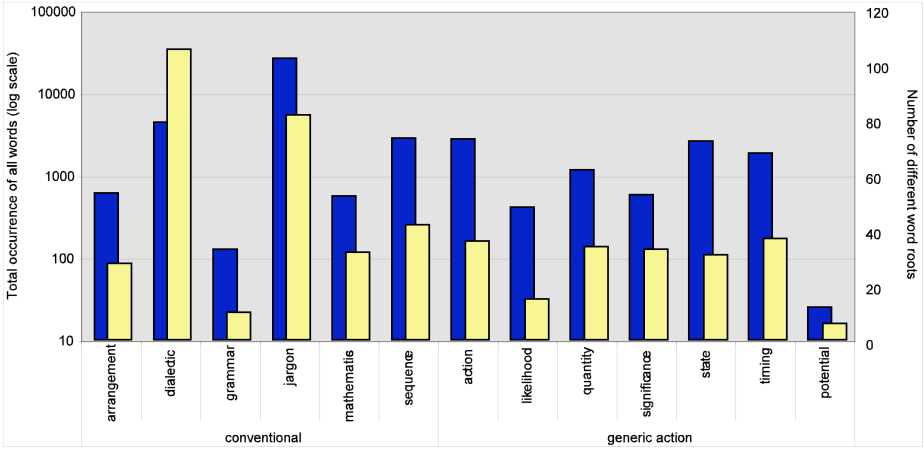

Fig. 2. Occurrences of conventional descriptions of computation and action

Most of the nouns that appeared in the sentences of JavaDoc were either identifiers or domain entities, neither of which were analysed (because, as explained, they do not reveal general concepts of program behaviour). The remaining nouns, after removing these two categories, were almost exclusively the highly conventional idiomatic jargon terms that might be found in a Java textbook. There are about 80 of these terms, and they are highly recognisable to any computer scientist: nouns such as

string, stream, type, object, instance, element, member, parameter, constant, event and exception (as well as some adjectives such as public, static and void). Some of these are keywords in Java, while some are keywords in other languages, that might easily have been chosen as keywords by the designer of Java. They are technical terms with precise definitions, and a documentation writer would have little choice other than to use the correct term. Less frequent, but also conventional, standardized jargon was the regular use of technical mathematical terms (e.g. function, factor, addition, enumerate).

In addition to programming and mathematical jargon, there are several other categories of vocabulary that are less specialized, but still highly conventional in computing discourse. It is quite conventional to describe algorithms as processing data structures that consist of sequences (e.g. insert, alternate, next), arrangements (e.g. pair, allocate, merge), or grammars (e.g. syntax, clause). These technical conventions are not analysed any further in this paper. Finally, I found frequent use of the dialectical terminology that is required whenever we describe complex logical or philosophical matters in English (e.g. explicit, example, otherwise, therefore). Although it is possible to analyse the underlying conceptual metaphors in philosophical discourse [6], the results do not have any specific bearing on mental models in programming, so I have grouped this category with mathematical and programming jargon as simply conventional terminology that we would expect to find in any corpus, and do not have particular bearing on the main hypotheses of this research.

4.2 Metaphors of Action

Much of the behaviour of software is described in terms of actions that are to be performed by components, or by the program as a whole. My analysis found many references to generic action (e.g. operate, perform, effect, use, activity), whose description might include the importance or significance of the action (e.g. typical, important, optional), the potential and likelihood with which it will occur (e.g. certain, likely, impossible), and the nature and consequences of the action in terms of generic change of state (e.g. terminate, new/old, modify, start, create). Overall, these can be described by the conceptual metaphor COMPONENTS ARE AGENTS OF ACTION IN A CAUSAL UNIVERSE.

The causal nature of software components and programs means that they have individual histories, and that their internal state is described as changing over the course of time (e.g. regular, recent, immediate) PROGRAMS OPERATE IN HISTORICAL TIME. Furthermore, the effects of action in the causal universe are discrete, able to be counted, compared and measured, thus requiring descriptions of quantity (e.g. large, less, many). PROGRAM STATE CAN BE MEASURED IN QUANTITATIVE TERMS.

Description of program behaviour in terms of action and change may seem unsurprising. Nevertheless, we should note that this was not a foregone conclusion. Internal program operation might easily have been described using conventions that are based on set operations or declarative constraints rather than actions, causality and state change. The object-oriented programming paradigm supports both imperative and declarative specification styles, and documentation writers might choose to define

behaviour in a declarative style. For example, Pane's study of natural specification style among children found regular use of declarative specifications of behaviour rather than descriptions of specific actions [10]. The frequency with which I found metaphors of action suggests that professional programmers may be better served by a combination of declarative and imperative specification styles.

4.3 Social Metaphors

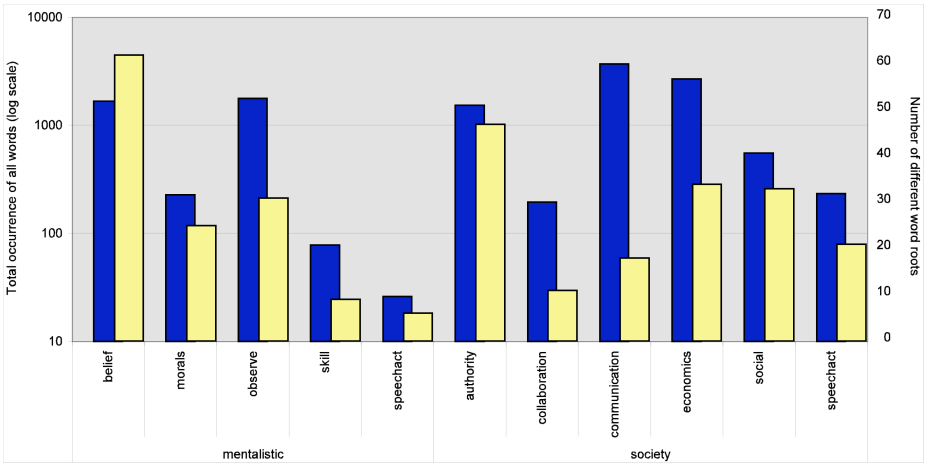

Fig. 3. **Occurrences of social and mental metaphors**

A surprising finding in early phases of my analysis was the occasions on which program behaviour was described in social terms (e.g. reconcile, collude, accompany). Classes and methods of the java libraries are described as associating and interacting with each other, as well as with new components that might be implemented by the person reading the documentation. The underlying conceptual metaphor can be described as COMPONENTS ARE MEMBERS OF A SOCIETY. The mechanisms of this society mirror and imitate much of the range of human society. There is some indication, for example, that components participate in democratic decision making, although this was described too infrequently to be a major finding. More significant findings were the many descriptions of economic activity (e.g. distribute, deliver, obtain) suggesting that COMPONENTS OWN AND TRADE DATA. The metaphorical society of software components is a highly structured one, and is described in terms of legal constraints and authority structures (e.g. impose, permit, contract, violate) so that COMPONENTS ARE SUBJECT TO LEGAL CONSTRAINTS.

Any society relies on communication between its members, and it seems that components share a wide range of human communicative behaviour. There is relatively frequent description of interaction between components as speech acts (e.g. instruct, query, offer, advise), suggesting the conceptual metaphor that METHOD CALLS ARE SPEECH ACTS. The nature and structure of the information communicated is

not limited to technical terminology, but uses a wide range of communicative styles (e.g. refer, describe, indicate, represent), such that we can say COMPONENTS HAVE COMMUNICATIVE INTENT.

4.4 Mentalistic Metaphors

When systems are partitioned into components, one of the main objectives of such partitioning is to achieve "information hiding". This was often described and interpreted by the javadoc authors as if the components are far more than containers for information, but are cognitive agents having their own beliefs and intentions. The information that a component has available to it is described in terms of knowledge and belief (e.g. interpret, consider, assume). On the basis of this knowledge, components are able to choose courses of action, but this too is described mentalistically (e.g. intend, desire). These suggest that A COMPONENT HAS BELIEFS AND INTENTIONS. Much of the activity of a component is concerned with gaining access to information from elsewhere. From the mentalistic perspective, this seems to be described in terms of observation (e.g. measure, observe, recognize, scan). COMPONENTS OBSERVE AND SEEK INFORMATION IN THE EXECUTION ENVIRONMENT. As agents with their own intentions, it seems that the action of a component can also be evaluated in terms that might normally be restricted to descriptions of human actors (e.g. fair, malevolent, graceful), such that COMPONENTS ARE SUBJECT TO MORAL AND AESTHETIC JUDGMENT.

4.5 Physical Metaphors

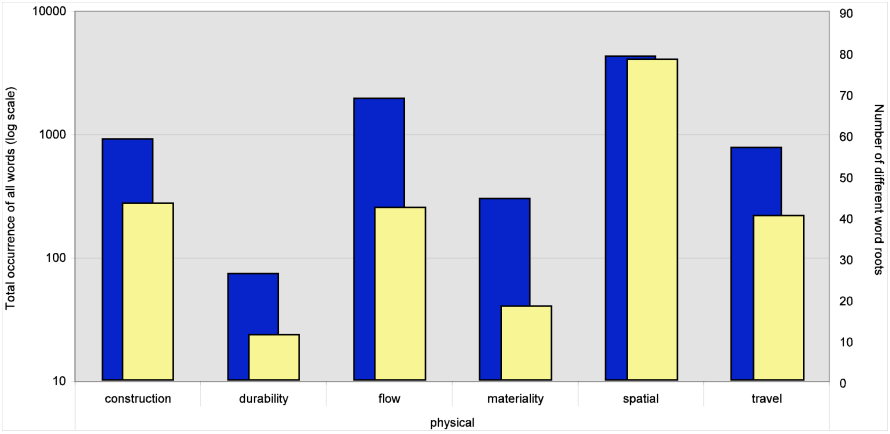

Fig. 4. **Occurrences of spatial and physical metaphors**

Previous research reported by Petre and Blackwell [11] revealed that expert programmers often conceive of their program structures in terms of visuospatial

images. That study focused on private experience, rather than descriptions that would ever be documented, or even described to another programmer. Nevertheless, the analysis of javadoc does reveal references to such spatial environments, even if the environments themselves are never explicitly described. Spatial relations occur regularly (e.g. back, contain, between, position), confirming the metaphor that PROGRAMS OPERATE IN A SPATIAL WORLD WITH CONTAINMENT AND EXTENT. Algorithms often involve some notional reference point traveling or moving about in this world (e.g. turn, ascend, flip), with descriptions such that EXECUTION IS A JOURNEY IN SOME LANDSCAPE.

Petre and Blackwell reported that designs can be experienced as buildings, or arrangements of structures and material within an artificial built environment. The metaphorical space of program execution is indeed described as a space of construction and physical mechanism (e.g. adjust, structure, form). These structures have physical and material properties (e.g. dynamic, efficient, hard), and they may be more or less durable, requiring intervention and maintenance (e.g. preserve, degrade, maintain). The overall metaphor is that PROGRAM LOGIC IS A PHYSICAL STRUCTURE, WITH MATERIAL PROPERTIES AND SUBJECT TO DECAY.

An interesting aspect of this environment is that data moves and is moved from one component to another, potentially corresponding to movement through the overall space. In addition to describing the physical paraphernalia of material flow (e.g. buckets, channels), the flows themselves are regularly described (e.g. fill, source, generate, empty). Within the metaphorically physical world of the program, DATA IS A SUBSTANCE THAT FLOWS AND IS STORED.

4.6 Metaphorical Borrowings

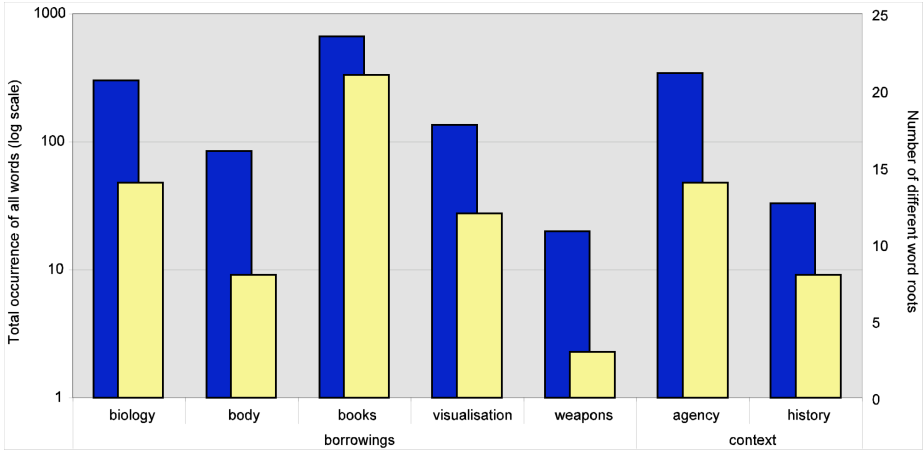

Fig. 5. Metaphors borrowed from other domains, and context of operation

In addition to the systematic metaphors described above, there are a number of more specialized views of computing, which draw on metaphors from other domains of knowledge. To some extent, these reflect the intellectual context in which computing is carried out. Early software engineering was conducted in the context of military funding and applications, and there are a few remaining metaphors that can be seen as drawing on the vocabulary of weaponry (e.g. target, trigger). These appear in other scientific domains also, suggesting a conceptual metaphor in which TECHNICAL RELATIONSHIPS ARE VIOLENT ENCOUNTERS. Alternatively, there are a set of conventions in which the data manipulated by a program is described in literary terms as a text (e.g. annotate, abbreviate, write), suggesting the conceptual metaphor that PROGRAMS CAN AUTHOR TEXTS. It is also possible to describe information constructs in non-textual, visual forms (e.g. render, exhibit, display), so that PROGRAMS CAN CONSTRUCT DISPLAYS. It is likely that new metaphors will continue to develop, adopting academic terminology current at any time. The rise of biosciences and bioinformatics is likely to result in increasing use of biological metaphors (e.g. mutate, clone, family, head/body/tail). In the broadest terms, we may find that DATA IS A GENETIC, METABOLIZING LIFEFORM WITH BODY PARTS.

4.7 Context of System Programming and Operation

The documentation of an application program is likely to be based mainly on descriptions of the domain in which the application operates. Internal operation of the program will interact with the requirements of that external world, so that conventional programming work involves maintenance and manipulation of both internal models and domain models. In the case of the Java libraries, there is no specific application domain, but only a highly generic execution context. Nevertheless, there are some interesting metaphors describing the nature of this generic context. One of these describes the sharing of agency between the software (e.g. automatic, code) and either application developers or application users (e.g. manual, human), applying the metaphor that SOFTWARE TASKS AND BEHAVIOUR ARE DELEGATED BY AUTOMATICITY.

The developers of the Java libraries are also highly aware of the context of technical evolution and standardization within which their work is situated. This is described in terms borrowed from human culture and history (e.g. legacy, traditional, obsolete), with the perhaps obvious metaphor that SOFTWARE EXISTS IN A CULTURAL/HISTORICAL CONTEXT.

5. Discussion

This research has revealed a number of systematic conceptual metaphors in the documentation of central Java libraries. Although the relative frequency of different categories is obviously influenced by the object-oriented programming paradigm, many of the metaphors found do not appear to be specific to OO libraries, or to Java, but reflect generic mental models of software operation. Some of them are already

familiar, because they are derived from the standard concepts and terminology of computer science and programming textbooks. I have not attempted to deconstruct that standard terminology, although this would certainly be an interesting exercise, perhaps based on a corpus of educational material. Many of the metaphors that I have identified by induction map onto certain perspectives in OO programming and design, for example the relationship between economic "ownership" of data that I have described, and the familiar concept of encapsulation, which is often described loosely as a matter of an object "owning" data. I have attempted to emphasise the external reference of the metaphor rather than the existing formalisation, but certainly recognize that they may appear very familiar to Java programmers.

Findings from this study that are worthy of note include the predominance of descriptions of causal action (rather than more declarative style specifications of component behaviour and interaction that might be supported within the OO paradigm), and confirmation of Petre and Blackwell's earlier findings regarding programmers' use of spatial and physical imagery [11]. The most interesting new finding is the predominance of cognitive and social descriptions of software components. The presumption of object-oriented languages, and indeed of packaged libraries, is that software should be assembled from such components. The traditional metaphorical view (e.g. [1]) is that these components should be viewed in mechanical terms. Although this study did find reference to physical mechanisms with material flow between them, social conventions and concepts seem to be at least as important as the common conceptual basis of programming work.

It is interesting to observe that the java.bean documentation included only about 10 occasions on which the writer directly addressed the reader (there may also have been similar occurrences in the java.util documentation, but the automated analysis technique did not preserve them). These were phrased as advice: "we recommend …", "if you want, then you can …", "you might want to check out …", "we advise …" and so on. It is perhaps surprising how seldom such phrasing occurs, given that javadoc is the primary source of information for java developers. It seems that the java package authors far more often allow their code to act as a social proxy for themselves, describing the preferences and requirements of the class that they have written, but not placing themselves in this relationship. (Of course educational literature and tutorial guides are far more likely to include direct advice to the reader, although less interesting as source data for mental model analysis, because less likely to have been written by the programmers themselves).

The most novel metaphor implied by these findings is that SOFTWARE COMPONENTS ARE SOCIAL PROXIES FOR THEIR AUTHORS. This is highly interesting, and warrants further investigation of the social psychology of library authorship. For example, in the case of metaphors of legal constraint, it is programmers who define those constraints and make the laws of the society in which their components act. The habitual use of legal terminology, political and business authority structures encodes hierarchies or power relations. These range from harsh (violate, constrain) to conciliatory (negotiate, elect). No doubt programmers and library developers are aware of such dynamics, although the way that they are revealed in the vocabulary of system documentation may be largely unconscious.

5.1 Weaknesses of the Method

Conceptual metaphor analysis is always a subjective exercise, as it relies on the re-interpretation of texts. In my own experience of the field, some research papers appear far more like works of literary criticism than empirical scientific analysis. There is some ground for empiricism in the quantitative comparison of frequency of occurrence, and this allows a degree of replicability, at least with respect to the generality of results, even if not their exact interpretation. In particular, the large scale quantitative analysis in the final phase of this study, although apparently substantive evidence, is also problematic. The quantitative information has enabled useful frequency comparisons, but the frequency of individual words taken out of context must be interpreted with care. In this study, the comparison to a prior manual coding phase was essential, and even then, it was possible to make errors. For example, an earlier version of this paper accidentally counted occurrences of the word "import" as referring to economic activity, when in fact I should have remembered that this is a Java keyword, and therefore fell into a group that would not be analysed as expressing further metaphors. A more rigorous study might employ dual coding and inter-rater reliability tests.

5.2 Implications for Design

This corpus analysis has found more diversity in conceptual metaphors of Java programming than might have been expected on the basis of "official" advice and jargon. Nevertheless, this diversity should not be surprising. Individual programmers will have differing habits of thought and preferences, and different applications and technical problems require different conceptual approaches. In the domain of software specification and design, the need to support diverse models has been recognized in the wide variety of different notational formalisms that were been brought together in the definition of the Unified Modeling Language (UML). UML thus helps programmers (and non-programmers) who think in different ways to work together. Different programming languages also cater for different conceptual models, but it is unusual to see a very wide range of programming paradigms used in a single commercial project, to the extent that is common in UML use.

Now that multi-language development and execution environments such as Eclipse and .NET are becoming widespread, it would be sensible to think more carefully about the conceptual models on which they are founded. In particular, the history of UML (incorporating earlier generations of notations from other paradigms alongside more recent object oriented concepts) suggests that it is both useful and possible to support a variety of conceptual design models, so long as these can be integrated via common interface semantics. In the case of programming languages, optional alternatives might include conventional declarative and imperative programming models, but could also allow for the some of the conceptual models revealed in this study that are not directly supported by conventional languages. Support for visuo-spatial imagery is one of these, and may well explain the persistent intuition that there is some advantage in visual programming languages. More significantly, support for the widespread social metaphors found in this study is absent both from contemporary

programming environments, and from programming language research. The use of social metaphors as a fundamental model of programming may be a productive direction for future research.

6. Future work

This analysis has been based on a single corpus, and the findings therefore describe mental models of programmers working with the Java language, and within the object-oriented programming paradigm. It seems likely, given the range of conceptual metaphors that have been found, that many of these will also be found among programmers working with other paradigms, although probably in different proportions and frequencies. Nevertheless, this should be confirmed by similar studies of corpora developed within those other paradigms. An obvious target would be the libraries of one of the major functional programming languages, such as Haskell or ML. Where these libraries include components intended for use by a larger community, will the documentation of those components also describe interaction in terms of material flows, and in terms of social relationships with the components constructed by others? It seems likely that they will, but this can only be determined by further study.

The method applied here has been an inductive one, and has not attempted to address the question of where these metaphors come from. Undoubtedly they will be reflected in textbooks, course material, online tutorials (including the overview "guide" material that is also included, alongside JavaDoc documentation, in the standard Java distribution), and professional resources such as documentation of advanced programming patterns. Analysis of these different sources would be a valuable complement to the present study. In particular, it would allow further inspection of my (problematic) claim that some jargon terms are so completely embedded in educational and professional discourse that they should not be analysed. In the present study, that particular decision was taken in order to focus on the identification of novel and unanticipated metaphors. In future, I suggest that only language keywords be excluded (and even these might be a focus of analysis in order more deeply to inspect the mental models of language designers).

Acknowledgements

William Billingsley and Darren Edge provided valuable assistance, criticism and suggestions in this work.

References

1. Blackwell, A.F. (1996). Metaphor or Analogy: How Should We See Programming Abstractions? In P. Vanneste, K. Bertels, B. De Decker & J.-M. Jaques (Eds.),