Title

Language shaping power: Bakhtin, Cassirer, and Phenomenology

By

Ivar Tormod Berg Ørstavik
The Faculty of Informatics and e-Learning
Sør-Trøndelag University College
7004 Trondheim
Norway

Email: ivar.orstavik@hist.no
Phone: +47 92030838

Abstract:
This article illustrates how morphology in programming languages influences interaction among programmers. To facilitate this illustration three theoretical approaches (Dialogism, Cassirer, and Phenomenology) are contrasted against each other. On this theoretical platform an experiment is then performed. This experiment shows how different morphological type systems in object-oriented programming encourage and discourage different programming practices and power structures. The study concludes that both the theoretical comparison and the experiment depict Dialogism as a uniquely socially oriented and down-to-earth theory of language.

Title

Language shaping power: Bakhtin, Cassirer, and Phenomenology

1. Introduction - An applied linguistic approach to theories of language

Applied linguistics combines theories of language with empirical data. These syntheses often focus on solving "real-world" problems and place linguistic theories in the background. Such an approach fits well with the name applied linguistics. But, this approach does not epitomize applied linguistics. Applied linguistics also foregrounds theories and uses empirical data as a background against which to investigate these theories. Hence, applied linguistics both solves problems *and* enhances linguistic theories by constantly flipping empirical data and theory as foreground and background.

This applied linguistic study aims at both solving problems related to wording inter-textual references in computer programs *and* understanding if and how Dialogism is particularly suited to such a task. To accomplish these goals Dialogic theory is first foregrounded against Cassirer's theory of language and then against a subjective Phenomenological perspective. These comparisons illustrate the complex link from Cassirer to Dialogism, the differences between Dialogism and a subjective Phenomenological tradition, and how these links and differences make Dialogic theory a unique analytical perspective.

In part two of this study, this sharpened Dialogical perspective functions as a platform on which to analyze and experiment with wording[1] inter-textual references in computer programs. This empirical investigation illustrates how programming language morphology directly shapes interactional patterns among programmers and how new morphological means can empower programmers so to enable new forms of interaction.

[1] Here meaning doing, making, and using words.

2. Cassirer's legacy in Dialogism

Tragicomical can be used to describe the link between Cassirer and Bakhtin's theories. Their common focus on laughter, comedy and carnival as philosophical subjects is a direct line between the two. In addition, the need for theoretical studies to uncover "Bakhtin's verbatim translation of over half a page of Cassirer's" (Poole 1998: 543) and similar references of Cassirer's line of thought in Bakhtin's works tragically exemplify how politics may influence research.

However, for me as an applied linguist, the most thought-provoking link from Cassirer to Bakhtin is the method of historical studies of texts and thought: "historicism" (ibid: 546f). This method uses textual data spanning longer historical periods to understand the emergence of linguistic phenomena, turning the focus from the linguistic structures themselves to their process of change (Bakhtin 1986: 65; Cassirer 2006: 91). Viewing language over time reveals processes of change, and it is through seeing these changes that we get a glimpse of how our thinking is connected with our language – ecologically.

Furthermore, this historical method has a second important characteristic. Textual data is selected using a wide conception of language. Cassirer (1953; 2006) sees language in relation to other symbolic forms such as technology, religion and myth. Bakhtin, on the other hand, stresses language variety: "Only with the coming of 'mnogoiazychie', a pluri-lingual environment (polyglossia), can 'consciousness be completely freed from the power of its own language and linguistic myths.'" (Bakhtin 1981: 61 cited by; Brandist 2004: 150). Hence, differences between and within a broad spectrum of languages are used to shed light on each other rather than working with examples of a coherent, unitary, normal language.

In the Scandinavian dialogic tradition, few studies use methods that enable neither a long historical perspective nor a polyglot language view. While the historical timeline and emergence in language is well described in theoretical models such as "diatope" (Evensen

2002) and “double dialogue” (Linell 1998), the associated methodological guidelines from Cassirer and Bakhtin are not as evident. When this theoretical and methodological synthesis is broken, discrepancies between theory and method might arise. First, macro-historical theoretical perspectives risk being set up against essentially micro-historical empirical data. Second, the exceptional and abnormal is often missing. Researchers need language differences to create meta-perspectives on their own languages and to understand how these languages shape their own language-based perceptions.

3. Dialogism inspired by the Soviet Union

The historical, polyglot method goes hand in hand with the theoretical stance Bakhtin and Cassirer take on inter-subjectivity. Common in their understanding of language is a view of inter-subjectivity rather than subjectivity as the driving force behind language evolution. However, on this issue Bakhtin was inspired by contemporary Soviet views on language development. Marxist literature “constitute the main source of Bakhtin’s sociological and historical account of language in his essays of the 1930s.” (Brandist 2004: 146) Cassirer, on the other hand, a Neo-Kantian, discussed his theories with other German philosophers heavily preoccupied with individual consciousness and subjectivity. These discussions led him to pursue an “Objectivity of the symbolic form” grounding inter-subjective processes (Cassirer, cited in Heidegger 1997: 205). Although both might conclude that inter-subjectivity drives language development, Bakhtin and Cassirer approach this idea from almost diametrical theoretical origins.

Bakhtin’s theories also differ from Cassirer’s in their down-to-earth approach to language. Voloshinov “translated inter-subjectivity into discursive forms: dialogic relations” (Brandist 2004: 146) and thus provided a more concrete approach to language. Combining a concrete approach to language with the theoretical concept of inter-subjectivity, the dialogic perspective enables researchers then and now to describe inter-subjective phenomena in empirical data from text and discourse.

Cassirer and Bakhtin share a historical, change-oriented method and the idea of language as basically inter-subjective. But the strong social and down-to-earth starting point in Bakhtin's theories seems political compared to Cassirer's philosophical and abstract starting point.

4. Dialogism vs. subjective Phenomenology: time and semantics

To compare Dialogism with a subjective Phenomenological perspective, this study will briefly discuss two topics: time and semantics. These discussions take a simplistic, birds-eye perspective from both a subjective and inter-subjective viewpoint. As such, these discussions exemplify how different theoretical preferences toward inter-subjectivity and subjectivity can affect the outcome of empirical analysis, and should not be read as in-depth analyses.

Time. Arguing against a tradition that uses eternity as its point of departure from which to understand time, Heidegger states that "time 'is' only for a being that lives with an awareness of its own mortality" (Alweiss 2002: 118). Heidegger "believes that we […] can only understand the phenomenon of time from our mortal or finite vantage point" (ibid.: 117), and since eternity cannot be conceptually grasped by such a being, eternity must be replaced by something that can: the subject's death. From this subjective, phenomenological viewpoint, "time" is thus tightly linked with the "individual subjective," both concepts informing the other.

However, an individual, subjective experience of death is no more within our human grasp than eternity: "death comes from outside and transforms us into the outside" (Satre, cited in Alweiss 2002: 118). This critique refers to the scope and limits of a subjective perspective on time. However, an inter-subjective approach to death is not open to the same criticism. Even though I cannot experience the phenomenon of my own death, and live to talk about it, I can experience quite tangibly the death of others. Individual death in general and my own death can therefore become graspable and meaningful from an inter-subjective

perspective: my understanding of my own time is a reflection of my perception of others' time, their birth and death.

Viewing time as an abstraction of other's time also highlights time's plurality. Using a metaphor from computer programming, our "environment" can be understood as "multi-threaded." Several independent timelines and "flows of control" run concurrently and across each other's paths. Such plurality is essential in a Dialogic view of language as a heteroglot sea of utterances and voices, each with its own time and direction, overlapping, interacting, and flowing in and out of each other.

Semantics. Focusing explicitly on the social character of language, Dialogism has developed several analytical concepts to illuminate the intricate, reciprocate relationships between words, our thinking, interaction, and culture. A central concept within these theories is the social dimension of language and words. "[Language] is populated – overpopulated – with the intentions of others" (Bakhtin 1981: 294), and when we express reality with words in any language, we use the voices of others in order to compose our own perspective on reality. Thus, our words, thoughts, and actions are composed as "refractions" (ibid.: 300, 276) or "polyphonies" (Bakhtin 1984) of others' words and voices.

When viewing language and our perception of the world primarily as social references, semantics is seen in a new light. Rather than categorizing semantic relationships between an individual's word and a physical world or a universal logic, the meaningful references in words become their "semantic" relationships between social actors and their texts, i.e., their inter-subjective role (figure 1). The meaning ascribed to words is not viewed primarily in categories such as "information" or "indices" to domain-specific logics or lexica; the meaning potential of words is viewed rather as social cues or voices of others. So, even though both approaches to semantics focus on "meaningful references of words," the two have completely different concepts of what "meaningful" means.

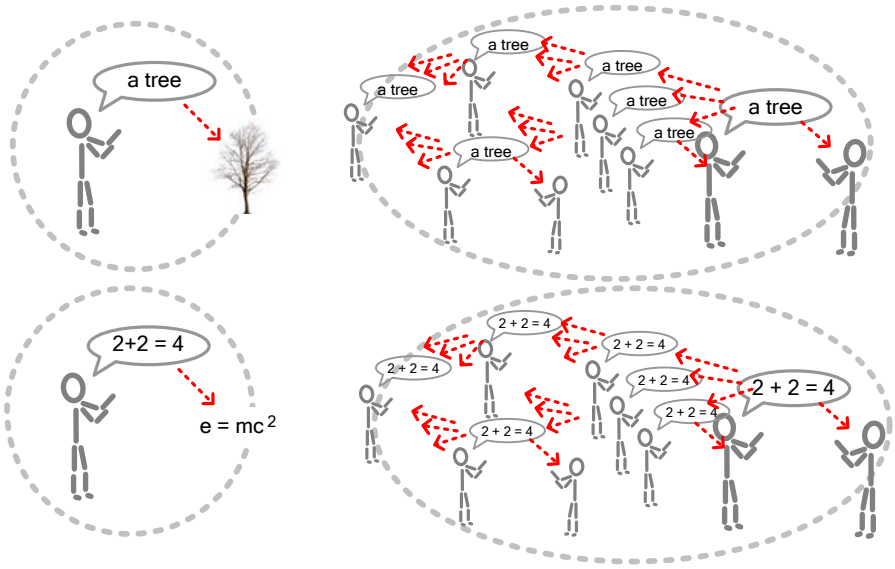

Figure 1: A subjective approach highlighting index and logic oriented references (arrows on the left) and an inter-subjective approach highlighting social, inter-textual references (arrows on the right).

When words are viewed as neither individual property nor asocial realities, social forces become a primary reality in language. Social structures are seen as within language itself, not as separate phenomena. In order to understand this social make-up of languages, it becomes necessary to describe the social dynamics and relationships inherent in language use.

5. Empirical analysis of words in programming languages

Dialogism stands out from related linguistic theories by focusing on socio-linguistic process combined with a historical, difference-oriented and down-to-earth approach. This study will now use this perspective to analyze computer program words and problems related to wording inter-textual references in computer programming.

Program texts in Java (Gosling 2000; Lindholm 1999) or Python (Python 2008) are composed of words (figure 2). These words can generally be categorized into four groups:

syntactic markers, primitives, variables and types. **Syntactic markers**, such as parenthesis, commas, and some so-called “reserved words”, reference generic functionality in special platform program texts such as a compiler or class loader. These words play primarily a syntactic role ordering the interplay between functionality described by other words. **Primitives**, such as numbers and built-in operators, function as atomic entities in the text. Both syntactic markers and primitives are morphologically closed and represent a finite set of symbols that can only be defined by the platform programmers. **Variables** function as pro-forms, i.e., placeholders for storing values or references. Variables are most often used “locally” within texts, but can also be used “globally” between texts. **Types** mark text as being or referring to entities such as a class or method. Types are the primary morphological means by which programmers link their texts to each other beyond the platform. Both variables and types are morphologically open, essentially enabling programmers to define and reference an infinite number of words. Even though all four groups enable inter-textual links, this study analyzes only *types*.

```java
import org.jdom.*;
public class HelloJDOM {
  public static void main(String[] args) {
    Element root = new Element("GREETING");
    root.addChild("Hello JDOM!");
    Document doc = new Document(root);
    org.jdom.output.XMLOutputter output = new org.jdom.output.XMLOutputter();
    try {
      output.output(doc, System.out);
    } catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

```python
import xml.dom.minidom

def printDocument(text):
    dom = xml.dom.minidom.parseString(text)
    print "<html>"
    for line in dom.getElementsByTagName("line"):
        print "<p>%s</p>" % getText(line)
    print "</html>"

def getText(node):
    text = node.childNodes[0]
    if text.nodeType == node.TEXT_NODE:
        return text.data
    return ""

document = "<text> <line>hello!</line> <line>and good bye..</line> </text>"
printDocument(document)
```

Figure 2: A Java text and a Python text. Syntactic markers and primitives are marked in **grey**, *variables in* italics *and types in* **bold**.

Types in Java and Python are relative to a path. A path, called “classpath” in Java and “pythonpath” in Python, is conceptually a list or cluster of program texts. In order to clarify which text entity that a type in another text entity refers to, the type reference is interpreted as within such a cluster of texts. Different path settings thus establish different inter-textual contexts enabling different interpretations of the same type reference. Only when program words are situated in inter-textual environments are concrete interpretations possible, and it is these interpretations that turn computer texts into computer programs.

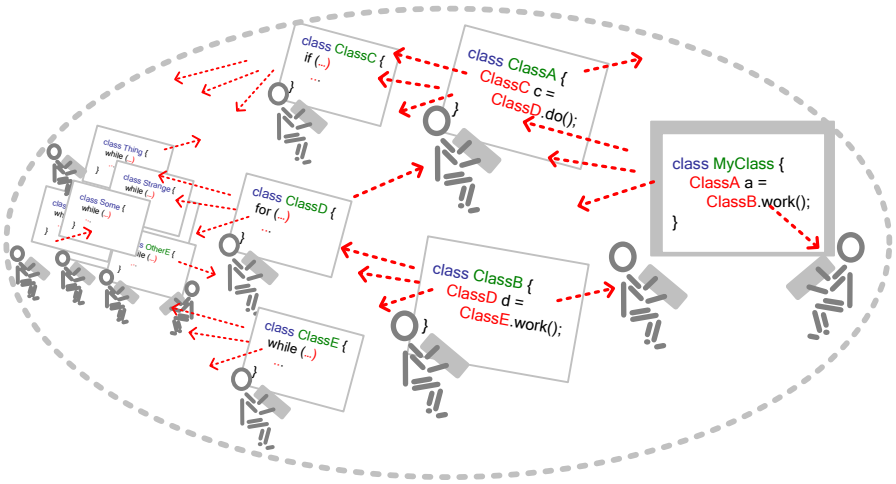

Figure 3: Words are interpreted in an inter-textual context.

In principle, program words such as types are therefore not specific to one particular interpretation: program words are not fixed indices to predetermined, universal functionality. What a type reference means must be determined within a path, and by changing this path a different interpretation of the same word is established.[2] Program words rely on each other in order to function, and program words can thus be viewed as words refracting and echoing each other across vast webs of words.

The value and usefulness of using the words of others cannot be overstated. First, the morphology, syntax, and pragmatics of types enable programmers to break down big problems into smaller, more manageable pieces. These pieces can in turn be solved with relative ease. Second, other programmers can reuse these solution pieces as simple words. By instantiating others' objects and invoking their methods thousands of programmers collaborate and link their program texts together. Type references thus enable programmers to build hugely complex systems by reusing fragments of code that other programmers

[2] Manipulating the path to creatively alter the interpretations of program words is a well-known method for hacking systems.

have spent countless hours developing, and types dwarf all other means of programmer collaboration.

Types enable programmers to literally construct their systems as refractions of other programmers' source code. Programs are composed as "polyphonies" whose meaning is not just an individually formed idea, but a mosaic of social collaboration in constantly evolving, rich programming language cultures.

6. Problems concerning the social interaction and dynamics of types

Using types also has its problems. The act of wording a type reference is not only to (a) explicitly formulate a type name; implied in this act is also the acts of (b) obtaining an actual program text that defines the type and (c) updating the path settings. Both Java and Python conceptually see both the type and paths as operating on a single computer or PC. On a PC, the act of wording a type reference typically compounds the acts of (a) writing a word, (b) finding and downloading a file with a corresponding class or method, and finally (c) updating the path settings on the PC so as to include this new file. Wording PC-based types in Java and Python is thus both a morphologic[3] and pragmatic practice.

Furthermore, using another programmer's word is not limited to single steps. A programmer can use another programmer's word that in turn uses a third programmer's word, etc. Thus, words create relationships that span several individual texts and individual actions. In computer programming these relationships are called "chains of dependencies," and a simple chain of dependencies example is presented below.

```
from minidomPrinter import printDocument
document2 = "<text> <line>hello again!</line> <line>ta ta..</line> </text>"
printDocument(document2)
```

Figure 4: A Python text: **printUser**.

[3] A set of conventions by which words can be composed.

Building on the **minidomPrinter** text (figure 2), a new program **printUser** is set up (figure 4). **printUser** directly refers to **printDocument**, and in order to function, the **printUser** text must be interpreted against a path that includes a text such as **minidomPrinter**. Without such an inter-textual context the **printDocument** word and thus the entire **printUser** text would fail to make sense. The new text **printUser** thus directly requires that we (b) find and download a file such as **minidomPrinter** and (c) update the pythonpath settings to include this file. There is a direct dependency between **printUser** and **minidomPrinter**. But, **minidomPrinter** in turn refers to yet another type **xml.dom.minidom**. In order for **minidomPrinter** to work, a text describing **xml.dom.minidom** must also be available within the path. Thus, an indirect dependency from **printUser** via **minidomPrinter** to **xml.dom.minidom** is established that forces the pythonpath around **printUser** and **minidomPrinter** to include **xml.dom.minidom**.

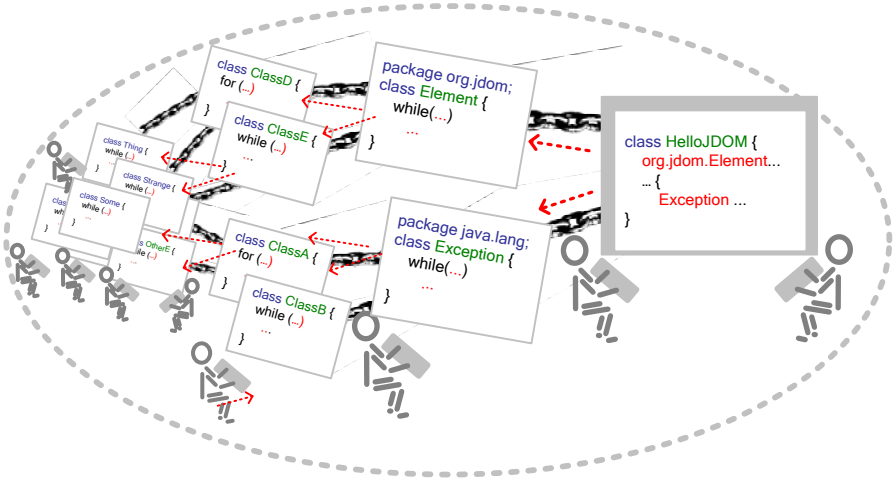

Figure 5: Chains of dependencies.

The practical, technical, social, and ethical concerns that follow in the wake of such dependency chains greatly influence programmers' choice of words.

First, chains of dependencies come into play both when programs are written *and* when programs run. Several mechanisms exist that allow programmers to, for instance, lump together all files in a path into a single file or semi-automate the process of downloading and updating other users' paths. However, all of these mechanisms essentially export the entire inter-textual sphere around a certain program text to every computer on which this text is interpreted, leaving the chain of dependencies intact. When such mechanisms are hindered by, for instance, legal terms or trust issues, the job of updating the inter-textual environment around a certain program file (tasks (b) and (c)) is wholly or partially transferred from the programmer to the users of his program.

Second, if a programmer chooses to use another programmer's word, he presupposes that his program will use the other programmer's text describing this word. But using this text can cost money and is often restricted by a legal license. As words and program texts depend on each other to function, these commercial and legal terms propagate up the chain, and this makes program words, in a very tangible sense, "populated with the intentions of others" (Bakhtin 1981: 294).

Third, when a programmer calls upon a second programmer's text, he also passes the control of his running program to the second programmer. As malicious or faulty pieces of code can cause serious problems, the use of other programmers' words always involves trusting them to a certain degree. However, when a programmer calls upon a second programmer that in turn calls upon yet a third, unknown programmer, the first programmer is required to trust not only the second, but also the third programmer. Thus, chains of dependencies come to constitute networks of trust, where the programmers maintain control over the inter-textual context collectively.

Individual programmers weigh these costs against technical benefits when choosing their words. If a word implies chains of dependencies that are either practically, ethically, or socially problematic, odds are that the programmer will choose another, less troublesome wording to address his needs.

7. Second and third party words in Java and Python

To lessen the burden of using types, Java and Python platforms are distributed with a standard library and a standard path setting. Standard libraries establish in practice a standard inter-textual context in a language: words within the standard library such as **xml.dom.minidom** and **java.lang.Exception** are, for all intents and purposes, always available. When using a standard library word, a programmer can therefore assume that the tasks of obtaining a file and updating the path settings are not needed.

The contrast between the ease of using standard library words and the potential problems of using other, non-standard words segregates words into second and third party. Second party words, such as **xml.dom.minidom**, can be used assuming no additional actions needed. Using a third party word, such as **org.jdom.Element**, a programmer must consider all the practical tasks and problems of obtaining files and updating path settings described above.

Furthermore, the choice between words not only considers existing, past program texts, but also potential, future program texts. Programmers anticipate other programmers referring to their program texts, and thus consider the potential practical, ethical, and social burden they transfer to other future programmers. In some genres such as program libraries, the program texts "response-ability" toward future users are as vividly apparent as their awareness of past, existing texts.

The social, practical and ethical concerns related to path-relativity of program texts that have been, are being, and might be written, creates a "centripetal force" (Bakhtin 1981: 270) that influences programmers' choice of words. The morphology and pragmatics of types in Java and Python combined with the platform distributors' ability to set up a default inter-textual context make programmers flock toward the standard libraries. While this excessive use of second party words might be harmless in itself, the monolithic grip

platform distributors get on program literature and the struggle third parties face getting their program words and voices heard is problematic.

8. Situated needs shaping actions - actions shaping morphology

Our informal experiment started with the development of a platform for programming multi-agent systems in Java (Ørstavik 2005; 2006). In our experiment we wanted different agent programs to interact with each other using the Java language as the medium while maintaining their ability to function autonomously.

The type morphology of Java quickly presented a problem when used as inter-agent medium. When one agent wanted another agent to respond to a request, they needed to involve words/types that were part of one agent's inter-textual context, but not the other's. There are many known mechanisms to work around this problem: the text describing the type could be passed directly between agents, or the agents could tell each other how to alter their path settings. However, such mechanisms provide alternatives to type morphology that make program texts heavy-handed to both read and write. Furthermore, the problem of extra-contextual types between agents is in principle the same problem of using third party types described above. A solution responding to both problems thus appeared beneficial.

To tackle this problem, we took a change-oriented linguistic approach to type morphology. Inspired by the pragmatics and morphological structures of the web, we extended the type morphology of Java to include URL-based types in a dialect called "theJ" (figure 6). The implementation of URL-types expands both the front- and back-end of the Java type system (Løkke 2005): front-end, type names can be written echoing complete URLs, and back-end, the compiler and platform is extended so as to locate, download, and utilize program files from the web automatically.

```
package thej;

import ftp://213.221.2.15/consult/EAMKA_02_05/client/lib/jdom-1.0.jar/org.jdom.*;
import ftp://213.221.2.15/consult/EAMKA_02_05/client/lib/jdom-
1.0.jar/org.jdom.input.SAXBuilder;

public class ExampleTwo {

    public static void main(String[] args) {
        http://the.hist.no/thej.ExampleOne parserA =
                        new http://the.hist.no/thej.ExampleOne("<b>", "</b>");
        SAXBuilder parserB = new SAXBuilder();
        try {
            parserB.build(args[0]);
        } catch (JDOMException e) {
            System.err.println(parserA.taggThis(e.getMessage()));
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Figure 6: A theJ text. Syntactic markers and primitives are marked in grey*, variables in* italics *and types in* **bold***.*

In theJ, programmers can refer to types outside the standard library as if they were second party words. The inter-textual context surrounding types is no longer an individually controlled PC-based path, but the entire web. The web infrastructure and URL morphology is merged into a new type morphology thus automating the tasks of obtaining files and updating path settings, even in chained dependencies, echoing “today's morphology is yesterday's syntax” (Givón 1971: 413 cited by; Hopper 1993: 26). While the web might be too wild an inter-textual context for several applications, the experiment illustrates directly how actions can and do shape our linguistic means.

9. Conclusion – empirical studies and Dialogic theory informing each other

Empirically, and in addition to solving a technical problem in inter-agent communication, theJ experiment creates a new, different morphological practice against which to compare an old, established morphological practice. By seeing the world of computer texts through the exotic lenses of URL-based, web-oriented types, the gravitational forces of PC-based, path-oriented types toward second party words at the expense of third party words become

evident. This new insight changes how we understand the other in programming and thus enables programmers to choose words and to interact with each other on different premises. Seeing this process from the inside as it unfolds has led me to conclude that programming "language exerts hidden powers, like a moon on the tide" (Brown 1989: 73), but that through individual actions we can change our languages so as to think of program words and act with each other in new ways.

Theoretically, this study explores both the historical and contemporary character of Dialogism. Dialogism was and is open to a polyglot linguistic environment, and it thrives on change and difference. Dialogism is also socially-oriented and down-to-earth: inter-subjective processes are linked with tangible language matter such as words and social interaction. Hence, both our empirical experiment and theoretical comparison suggest that Dialogism provides a unique analytical perspective that connects language and social interaction.

References

Alweiss, L. 2002. Heidegger and 'the concept of time'. *History of the Human Sciences* **15**(3), 117-132.

Bakhtin, M. 1981. *The Dialogic Imagination*. Austin: University of Texas Press.

Bakhtin, M. 1984. *Problems of Dostoevsky's poetics*. Minneapolis: University of Minnesota Press.

Bakhtin, M. 1986. The problem of speech genres. In C. Emerson and M. Holquist (eds.) *Speech Genres and Other Late Essays* (pp. 60-102). Austin: University of Texas Press.

Brandist, C. 2004. Mikhail Bakhtin and early Soviet sociolinguistics. In *Proceedings XI International Bakhtin Conference*, (pp. 145-153). Universidade Federal do Paraná , Curitiba (Brazil)

Brown, R. 1989. *Starting from scratch*. New York: Bantam Books

Cassirer, E. 1953. *The philosophy of symbolic forms: Language*. New Haven: Yale University Press.

Cassirer, E. 2006. Form og teknikk. In (eds.) *Form og teknikk: utvalgte tekster* (pp. 87-142). Oslo: Cappelen Akademisk Forlag.

Evensen, L. 2002. Convention from Below: Negotiating Interaction and Culture in Argumentative Writing. *Written Communication* **19**(3), 382-413.

Givón, T. 1971. Historical syntax and synchronic morphology: An archaeologist's field trip. *Chicago Linguistic Society* **7**, 394-415.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000. *The Java Language Specification*. Boston: Addison-Wesley.

Heidegger, M. 1997. Appendix IV: Davos disputation between Ernst Cassirer and Martin Heidegger. In R. Taft (eds.) *Kant and the Problem of Metaphysics* (pp. 193-207). Bloomington: Indiana University Press.

Hopper, P., and Traugott, E. 1993. *Grammaticalization*. Cambridge: Cambridge University Press.

Lindholm, T., and Yellin, F. 1999. *The Java Virtual Machine Specification*. Boston: Addison-Wesley.

Linell, P. 1998. *Approaching dialogue: talk, interaction and contexts in dialogical perspectives*. Amsterdam: John Benjamins.

Løkke, J. 2005. Implementing URL-types in Java. Trondheim: HiST.

Ørstavik, I. 2005. Designing multi agent systems using Java as an Inter-Program language. *JavaZone*. Oslo.

Ørstavik, I. 2006. Site of theJ programming language. Retrieved 1.6.2006 from http://the.hist.no.

Poole, B. 1998. Bakhtin and Cassirer: The Philosophical Origins of Bakhtin's Carnival Messianism. *South Atlantic Quarterly* **97**(3), 537-578.

Python 2008. Python Programming Language. Retrieved 17.9.2008 from http://python.org/.