

Fernuniversität in Hagen  
Fakultät für Mathematik und Informatik  
Lehrgebiet Programmiersysteme

**Bachelor's Thesis in Computer Science**

# **Shaping Statically Resolved Indirect Anaphora for Naturalistic Programming**

**A transfer from cognitive linguistics to the Java programming language**

Sebastian Lohmeier  
sl@monochromata.de

May 11, 2011

Supervisor: Dipl.-Inf. Andreas Thies  
Examiner: Prof. Dr. Friedrich Steimann



This work is licensed under the Creative Commons Attribution 3.0 Germany License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/3.0/de/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

For convenient searching and e-reading, an electronic version of this work is available at [http://www.monochromata.de/bachelor\\_thesis/index.html](http://www.monochromata.de/bachelor_thesis/index.html). The source code of the implementation described in later parts of this work can also be found there.

# Contents

<b>List of Future Work</b>	<b>v</b>
<b>Preface</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Related Work . . . . .	2
1.2 Aim . . . . .	3
1.3 Organization . . . . .	4
<b>2 Reference in Natural Languages</b>	<b>5</b>
2.1 Reference, Names, Deixis and Anaphora . . . . .	5
2.2 Common and proper names . . . . .	6
2.3 Direct Anaphora . . . . .	7
2.3.1 Pronominal anaphora . . . . .	7
2.3.2 Ellipsis . . . . .	8
2.3.3 Definite descriptions . . . . .	9
2.4 Cognitive Foundations . . . . .	9
2.4.1 Mental Representations . . . . .	10
2.4.2 Text-world models . . . . .	10
2.4.3 Focus and Activity . . . . .	11
2.5 Indirect Anaphora . . . . .	12
2.5.1 Anchoring based on thematic roles . . . . .	13
2.5.2 Meronymy-based anchoring . . . . .	14
2.5.3 Schema-based anchoring . . . . .	16
2.5.4 Inference-based anchoring . . . . .	16
2.5.5 Anchoring of Indirect Anaphors . . . . .	17
2.6 Summary . . . . .	18
<b>3 The Relations Between Natural Languages and Programming Languages</b>	<b>19</b>
3.1 Programming Languages considered Languages . . . . .	19
3.2 Naturalistic Programming Languages . . . . .	19
3.3 Summary . . . . .	20
<b>4 Reference in Java</b>	<b>21</b>
4.1 Names . . . . .	21
4.2 Deixis . . . . .	22
4.3 Zero anaphors . . . . .	23

*Contents*

4.4	Requirements for Indirect Anaphora in Java . . . . .	23
4.5	Summary . . . . .	24
<b>5</b>	<b>Indirect Anaphora for Java</b>	<b>25</b>
5.1	Constructing a Metaphor . . . . .	25
5.1.1	Pragmatics . . . . .	27
5.1.2	Syntax . . . . .	27
5.1.3	Cognitive Foundations . . . . .	28
5.1.4	Semantics . . . . .	29
5.2	General properties of indirect anaphora in Java . . . . .	32
5.2.1	Refential ambiguity . . . . .	32
5.2.2	Multiple anaphors per anchor . . . . .	32
5.2.3	Indirect anaphors as qualifiers . . . . .	32
5.2.4	Preconditions . . . . .	32
5.3	Anchoring based on the headers of invocables . . . . .	32
5.3.1	Preconditions . . . . .	33
5.3.2	Anchoring algorithm . . . . .	33
5.4	Anchoring based on fields and accessors defined by types . . . . .	34
5.4.1	Accessors . . . . .	35
5.4.2	Preconditions . . . . .	35
5.4.3	Anchoring algorithm . . . . .	36
5.5	Inference-based anchoring . . . . .	37
5.6	Test case nomenclature . . . . .	37
<b>6</b>	<b>Summary and Conclusions</b>	<b>39</b>
	<b>Bibliography</b>	<b>41</b>

## List of Future Work

2.1	Deixis and anaphora . . . . .	6
2.2	Include direct anaphora, based on Schwarz-Friesel's model . . . . .	9
2.3	Depth of conceptual decomposition . . . . .	10
2.4	Instantiation of concepts and specification . . . . .	11
2.5	Details on thematic progression . . . . .	18
3.1	Criticizing the concept of <i>natural language</i> . . . . .	20
5.1	Scripts in Java . . . . .	29
5.2	Underspecification of arguments . . . . .	34

*List of Future Work*

# Preface

This work grew out of the idea of getting closer to programming in natural language without appearing insane. That thought was motivated by the fact that specifications are often written in natural language (if a specification is written at all) and mention of the Metafor [LL05] system in the press, a couple of years ago.

Working on this thesis, I got in contact with a lot of topics not treated in a computer science major. Alas, the minor of the program I was on did not leave room to study fields not prescribed. This may be common to or even be the nature of minors, but my studies would have profited had the curriculum been less ISDN-like and been more like the Internet instead (in terms of openness, not speed).

Putting this idea aside, I acknowledge that the programming systems chair granted me all freedoms possible while I was writing my thesis. Its members have been very kind and helpful in supporting my thesis but also during the courses I had had with them before. I am grateful to Andreas Thies who supervised my thesis, as well as Daniela Keller, Christian Kolle and chairman Friedrich Steimann who participated when I outlined early results of my work and gave as much feedback as was possible at that time. Special thanks go to Roman Knöll of Universität Darmstadt, who sent me the results of the Pegasus Project's work on naturalistic references and took the time to answer my question on his work. I thank Susan Segebard who reviewed parts of this thesis. I am also indebted to my mother who supported me financially at the time when I wrote this thesis and tolerated my limited mood during that time.

Finally some notes on wording: although it is common in scientific texts that the author refers to himself as *we*, I will not do so in order to not conceal the fact that I had to write the thesis myself. In a similar attempt to avoid confusion I will use the pronoun *one* when referring to any third person instead of the commonly used *you* that would only be used to address the reader.

*Preface*

# 1 Introduction

The basic concept of allowing a person to communicate with a computer in his natural language will surely take many many years, and may exceed the lifetime of some of us. This does not mean that it is not a goal worth striving for.

These closing words from Jean E. Sammet's 1965 talk [Sam66] will, out of context, normally not be doubted. If one adds the title of the talk: *The use of English as a Programming Language* it can be expected that computer scientists shiver. Some might do because it is a goal still far from being reached and it is not clear how it could be done. Others might shiver in anger because they doubt that the goal could be worth striving for. This thesis serves to calm in both respects – not by presenting a ready-to-use solution, but by providing a tangible step into the direction of that goal.

In her talk, Sammet lays out two kinds of approaches to get to programming in natural language: top-down approaches that depart from natural language and attempt to accept syntactically unrestricted input with only a certain rate of successfully interpreted utterances that is to be improved in the course of development. Alternatively, bottom-up approaches depart from programming languages and guarantee the correct interpretation of all utterances while aiming at advancing the language over time to get closer to a natural language.

Early implementations of the former approach were actually a mix of both approaches in that they only recognized natural language utterances that complied on the basis of a restricted grammar, laid within a limited semantic domain and were still prone to error. In the *Natural Language* chapter of his book *Software Psychology*, Ben Shneiderman provides an overview and evaluation of natural language systems up to the end of the 1970's [Shn80, 198–213] that makes clear how verbose some of these systems had been. Regular users must have been annoyed at that over time. Shneiderman also points out that *proactive inference* can make these systems difficult to use: their similarity to English makes it hard to recall which subset of the English grammar they recognize, leading to errors in writing with these systems while the texts were easy to read and comprehend<sup>1</sup>. That natural language complicates the use and development of computer programs is also the core of Dijkstra's criticism [Dij78].

The programming language COBOL, in whose construction Sammet took part, may be regarded an instance of the bottom-up approach that tries to mimic English through keywords and word order but does not adhere to the grammar of English. Bryan Higman criticizes COBOL for this very feature [Hig67, 144], but adds that pronouns known from natural languages provide a way to shorten utterances in programming languages as had already been proposed for the programming language ALGOL [Hil65, 71-2].

---

<sup>1</sup>William Cook reported the same for AppleScript in 2007 [Coo07, 1-20] even though he noted that the project had been rather constrained. It might be that longer-lasting, well staffed projects are more successful in enabling users to program in natural language.

## 1 Introduction

Higman notes that it is not trivial to determine what a pronoun (that can function anaphorically, see below) refers to in natural language. While anaphora resolution<sup>2</sup> is still non-trivial, it is better understood by linguists these days. There have thus been recurring proposals to include more forms of anaphora in programming languages (see *Related Work* below).

It should be noted that recently the term *naturalistic programming (NP)* has been introduced while *programming in natural language* or *natural language programming (NLP)* have been used before. Naturalistic programming indicates a way of programming that is rooted in the bottom-up approach proposed by Sammet. The term will be further discussed in section 3.2.

My step towards the goal of programming in natural language can be clarified here already: I will use a bottom-up approach by adding forms of anaphora to Java. This does of course relate to programming in English in the same way that early rocket science is related to flying to Mars: the small steps are motivated by the bigger goal and while it remains unclear how far away the actual goal is, the value that the immediate steps provide for themselves gains relevance.

### 1.1 Related Work

This thesis was motivated by the broadly related work mentioned in the previous section. Work from the domain of linguistics that I will refer to in chapter 2 provides a basis for this thesis. The work listed in this section I consider closely related. I.e. the works listed seek to bring use of programming languages closer to the use of natural languages by proposing or implementing means of reference known from natural languages for programming languages<sup>3</sup>.

After excluding most related work, two projects remain that have influenced this thesis, one merely pointing out the field to be worked on and the other one doing early work in the field. In both publications *nature* and *intuition* could be referred to more critically and, due to the early nature of the research, both leave room for backing from cognition, linguistics and philosophy of language as well as empirical evaluation of results.

Lopes et al. [LDLL03] coined the term *naturalistic programming* and pointed out that current programming languages support only a limited number of kinds of anaphora, most of which are said to be structural, while some limited forms of temporal anaphora are said to have been introduced by aspect-oriented programming (AOP). They propose that more kinds of anaphora be added to programming languages, leading to *naturalistic programming* that they distinguish from programming in natural language and end-user programming. Since they mainly aim at re-generating interest in the topic, they include an extensive overview of related work, e.g. research in cognition, cognitive semantics and metaphors as part of terminology used in computer science. Their account of anaphora in linguistics, however, is syntax-heavy.

---

<sup>2</sup>Anaphora as used within this work corresponds to the linguistic term: an anaphora is a relation between an item in a text (called *anaphor*) and a previously mentioned item (named *antecedent* or *anchor*) by which the antecedent contributes to the meaning of the anaphor. An anaphor hints at its potential antecedent (*presupposes* it in linguistic terms). The process of locating the actual antecedent presupposed by an anaphor is called *anaphora resolution*.

<sup>3</sup>There is actually a gap between the work I consider closely related and the other works I reference. A lot of literature is available that fits into this gap. I am aware of that literature, but was not able to consider it due to time constraints. Topics that fall into the gap include end-user programming, meta-, literary-, aspect-oriented- and domain-specific programming and alternative means of method invocation.

The Pegasus project [KM06] developed a run-time model that is applied to natural language programming. [Hen08] extends Pegasus by analyzing existing means of reference in programming languages and proposes and implements a number of new dynamically-resolved reference mechanisms based on means of reference in natural language for what appears to be a subset of English. The types of reference are quite diverse and feature quantifiers and attributes, indirect anaphora are, however, not implemented. [Sta09] transfers these dynamic references to a modified version of Java called Rava by connecting the run-time model of Pegasus and the Java Virtual Machine (JVM) making Rava a naturalistic programming language. Resolution of references is based on a history list that contains potential antecedents, sorted in the order of appearance. The impact of control structures on referencing is not treated and the three works on Pegasus do not draw parallels to cognitive linguistics. The Pegasus project is still active: Roman Knöll is working on his dissertation, that will, besides other more practical results, include a detailed discussion of the term *naturalistic programming* which is highly desirable but yet outstanding in the current literature.

## 1.2 Aim

Although the idea of programming in natural language or anything closer to it than current programming languages motivated the thesis, I used the related works as a guidance to narrow down the topic to statically resolved indirect anaphora to have a topic of manageable size, i.e. to be able to yield concrete and novel results. The search for literature on linguistics further revealed a model from cognitive linguistics proved to be central for my work.

In general, I consider this thesis a discovery of unsettled territory. I will look for practical applications of indirect anaphora in order to verify the theoretical transfer of the concept of indirect anaphora. Because the way towards these applications is integral to this thesis, new issues raised are considered part of the outcome of the thesis. Some of these issues they have been highlighted throughout the text in boxes labeled *future work* that are indexed on page v to make it easier for readers of this thesis who want to work on the same topic to figure out an own starting point.

The following aspects are crucial to this work:

- Indirect anaphora were chosen to be implemented.
- A bottom-up approach is adopted by basing the implementation on an existing programming language (Java) and considering the impact that the complexity of this language has on the concept of indirect anaphora.
- Indirect anaphora in Java will be resolved at compile-time to exploit information easily accessible within the abstract syntax tree of the compiler.
- An existing cognitive model was found in the literature and will be used as the basis of the implementation.
- The nature of the relation between natural languages and programming languages will be discussed for the anticipated transfer will happen along this relation.

## 1.3 Organization

This thesis is interdisciplinary in that it applies cognitive and linguistic concepts to computer science. The chapters reflect the interdisciplinarity in that they represent a gradual transition from cognition and linguistics to computer science in their order of appearance.

The second chapter defines basic terms related to reference and anaphora, introduces the concepts of anaphora and indirect anaphora. Cognitive models will be described that are required to understand how humans resolve anaphora. The models are applied to text samples to detail the resolution of different forms of indirect anaphora.

The fact that natural languages and programming languages are called *languages* but attempts to program in English failed motivated chapter 3 that outlines the relations between natural languages and programming languages and discusses the term *naturalistic programming* in the context of these relations.

In the fourth chapter I analyze existing means of reference implemented in the Java programming language using the terminology from chapter 2 and develop requirements for indirect anaphors in Java.

In the fifth chapter, indirect anaphora will be transferred to Java.

The work closes with a summary and conclusions in the last chapters.

## 2 Reference in Natural Languages

Reference is an integral part of natural language<sup>1</sup>. Within the field of linguistics, semantics and pragmatics deal with reference extensively because reference constitutes meaning. Cognitive science comes into play when attempts are made to explain how readers resolve reference. Syntax plays a role in reference as well, especially by restricting possible reference, but indirect anaphora, the main form of reference in this work, is relatively independent of syntax which is why syntax is not devoted special attention in this chapter.

Linguists do not only deal with language in an abstract sense, but also concrete instances of language. Semanticists do e.g. treat forms ranging from parts of words to texts. Linguistic models of text are often not restricted to writing but can cover speech as well. Thus, a text can be characterized as a larger coherent utterance. When I use the terms *reader*, *writer*, *to read* or *to write* this does not mean that the model underlying the discussion would necessarily differ were a *listener* and a *speaker* involved instead.

While this work is focused on indirect anaphora, it is necessary to delineate indirect anaphora from other means of reference. For natural languages this can be done by quoting the literature as part of this chapter, for programming languages I discuss this matter using the example of Java in chapter 4. To maintain a close relation between natural language and programming languages, all samples in this chapter have been taken from the Java Language Specification [GJSB05]. This choice of a specification from the domain of computer science in preference to texts portraying everyday life typically found in linguistics is an explicit one. It is due to the (unproven) assumption that there could be a relevant difference between the use of reference in specifications and the use of reference in non-technical texts. I suppose that if a kind of reference is used in specifications written in natural language, then this kind of reference could also be useful in the implementation of the specification written in a programming language.

This chapter starts with an introduction to reference in general and explains means of direct anaphora before cognitive foundations are introduced for the discussion of indirect anaphora central to this chapter.

### 2.1 Reference, Names, Deixis and Anaphora

Attempts to discuss reference can be quite informal, starting with the "action of picking out or identifying with words" [Sae03, 23]<sup>2,3</sup>. For this work it is important to restrict reference to

---

<sup>1</sup>Although the examples given in this chapter are given in English, the concepts they illustrate can be found in other languages as well.

<sup>2</sup>Reference in linguistics does typically not include explicit references like inter-textual pointers or references to the bibliography of technical texts, its index or cross-references.

<sup>3</sup>Definitions of *reference* can also be quite complex, as in the case of Consten's definition [Con04, 56] that is reader-centric, process-oriented and cognitive but would be too detailed for the purpose of this work.

## 2 Reference in Natural Languages

a relation between linguistic expressions and extra-linguistic entities (see [Sch00, 22]). To be able to identify the entities participating in reference, it is then sufficient to fix that *reference* is a process in which a reader establishes a relation from a referring linguistic item towards a non-linguistic referent, e.g. by using a name in a sentence talking about a person that is referred to by the name.

Three terms are frequently used in linguistics when reference is talked about: *names*, *deixis* and *anaphors*. Names will be treated in the next section. A rough idea of deixis and anaphors is that both are linguistic expressions and that the former refers to sensually perceivable referents and the latter relate within the text (see [Con04, 6] with different terminology). Referring to the reader of a text using the word *you* is a form of deixis, but relating to a character in a novel that had been introduced in the prior text is anaphoric.

**Future Work 2.1 (Deixis and anaphora)** *Consten provides an overview of the history of the two terms deixis and anaphora and how they have been related to each other (in all possible constellations) [Con04]. Attempts to delineate the two terms have often been problematic, e.g. when dealing with reference to abstract entities or fantasy. Triggered by the observation that some words can establish both anaphora and deixis and the user cannot have a model for either anaphora or deixis exclusively, Consten created a model that integrates both anaphora and deixis and includes the reader's gradual distinction between the two. Following Schwarz-Friesel's work [Sch00] on domain-based anaphora, Consten developed a general theory of domain-based means of reference.*

*In this work I do not differentiate between anaphora and deixis in detail. When explaining natural language background, I will detail anaphora only. It would be interesting to treat deixis as well, though. Consten's work seems to be a good starting point for that.*

Whether deixis can occur in the source code of computer programs will be contemplated in section 4.2, names and anaphora will now be looked at.

## 2.2 Common and proper names

There are basic definitions of names like: "Names after all are labels for people, places, etc. and often seem to have little other meaning." [Sae03, 27] that are handy, or more differentiated ones, like van the Langendonck's [Lan07, 87ff.] that will be of use here. Important in the context of this work is that van Langendonck specifies that a proper name (also: proper noun) (1) refers to a unique entity, that is (2) highlighted within a class of entities by being given a name, (3) the meaning of the name does not (anymore) determine what the name refers to.

Presenting a definition of the term *name* and one of the term *proper name* raises the question what other names there are besides proper names. *Common names* (also: *appellatives*) are another kind of name. A common name is used for a class of entities or the entities of the class, for which only point (1) of the definition of proper names is valid. According to van Langendonck, the referent of a common name must actually comply to the properties required by the name [Lan07, 90].

Before a reader can resolve the referent of a name, she needs to be aware of the relation between name and referent, as in the following example.

**Sample 2.1** *If you like C, we think you will like the Java programming language.* ([GJSB05, xxv], emphases mine)

In the example *C* and *the Java programming language* are deictic because they refer to the two well-known programming languages that haven't been introduced in the text prior to the example. It is, however, possible to introduce new names before use in a text so that subsequent uses can be regarded as establishing an anaphoric relation (see [Lan07, 182] [Mit02, 8]).

## 2.3 Direct Anaphora

While the meaning of proper names is detached from what they refer to, the meaning of a phrase used anaphorically is tightly connected to the referent of that phrase. Among the typical classifications of kinds of anaphora is the division into direct and indirect anaphora. Schwarz-Friesel characterizes direct anaphora as follows. The most important function of an anaphor is to refer back to an antecedent in the previous text in order to draw its meaning from the relation to it. Anaphor and antecedent *can* be co-referential (i.e. refer to the exact same referent) and anaphors can maintain the topic of the text (thematization) or shift it by introducing new information (rhematization). Understanding anaphors is a cognitive process [Sch00, 64f.]. If both anaphor and its antecedent that are given in the text, the anaphor is called *direct anaphor* and its relation to the antecedent is called *direct anaphora*. If the referent of the anaphor is not given in the text, but closely related to a so-called anchor which is given in the text, the anaphor is an *indirect anaphor*; the relation between *indirect anaphor* and its *anchor* is called *indirect anaphora* (see below). Direct and indirect anaphora do not form a dichotomy, though. Schwarz-Friesel showed instead that they are two extremes of a gradual concept of anaphora. It is, however, true for both direct and indirect anaphora that "one of the properties and advantages of anaphora is its ability to reduce the amount of information to be presented via abbreviated linguistic forms" [Mit02, 12].

Schwarz-Friesel [Sch00, 59ff.] highlights what she calls *canonical conditions* for the relation between anaphor and antecedent, i.e. prototypical rules that will not be met by exceptional cases: (1) that gender and number of anaphor and antecedent agree, (2) anaphor and antecedent are semantically equivalent or at least compatible and (3) anaphor and antecedent are reasonably close so that continuity of the textual reference is maintained.

Pronominal anaphora and zero anaphora are kinds of direct anaphora and the linguistic forms used to realize them occur in programming languages as well (see chapter 4). Definite descriptions can also be used as direct anaphors and could potentially be useful in programming. The following sections contain brief outlines of all three kinds.

### 2.3.1 Pronominal anaphora

The use of pronouns like *he*, *her*, *it*, *himself* as anaphors is the most common one in introductory discussions of anaphora. An example is given below<sup>4</sup>.

---

<sup>4</sup>I added subscripted numbers in the example to express that all phrases indexed with the same number share a referent.

## 2 Reference in Natural Languages

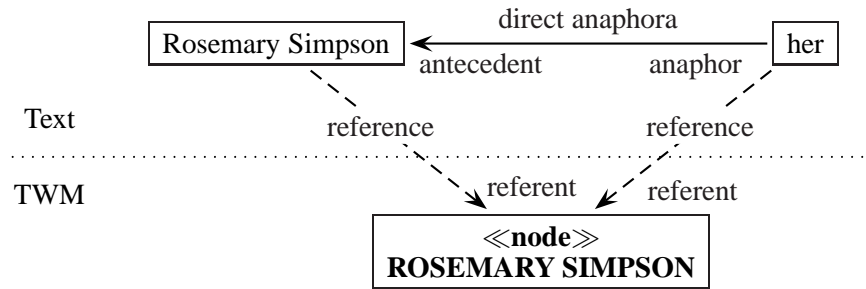


Figure 2.1: Relations in text and text-world model for sample 2.2

**Sample 2.2** *Rosemary Simpson<sub>1</sub> worked hard, on a very tight schedule<sub>2</sub>, to create the index<sub>3</sub>. We<sub>4</sub> got into the act<sub>5</sub> at the last minute<sub>6</sub>, however; blame us<sub>4</sub> and not her<sub>1</sub> for any jokes<sub>7</sub> you<sub>8</sub> may find hidden therein<sub>3</sub>. ([GJSB05, xxv], emphases mine)*

The antecedent of *her* is clearly *Rosemary Simpson* because in this sample the referent of this name is the only referent representing a singular female person. Figure 2.1 depicts the direct anaphora relation between *her* and *Rosemary Simpson*, illustrates that anaphora is a relation within a text contrary to reference that connects phrases of the text to nodes in a text-world model (TWM) constructed by the reader (see below). The co-referentiality of antecedent and anchor becomes clear as well. Note also that in the sample text *we* and *you* are deictic, but *therein* refers to *the index* which itself is actually an indirect anaphor that can be resolved without the presence of an antecedent because the previous text concerns the authoring of the specification.

### 2.3.2 Ellipsis

In certain syntactical positions items can be removed from a sentence without hampering the understanding of the sentence. The resulting *ellipses* (depicted as  $\emptyset$ ) are also called *zero anaphor* due to the fact that a plausible interpretation of the sentence is constructed by filling the empty position with an antecedent [Mit02, 12]. Ellipsis can, however, also be used deictically (see [HH76, 144]). Among the items that can be removed from a sentence are pronouns:

**Sample 2.3** *If an eligible \ is not followed by u, then it is treated as a RawInputCharacter and  $\emptyset$  remains part of the escaped Unicode stream. ([GJSB05, 15],  $\emptyset$  mine)*

Zero pronouns do not work in all syntactical positions, though (the asterisk in front of the sentence marks it as invalid), as can be seen from a modified version of the last sample:

**Sample 2.4** *\*If an eligible \ is not followed by u, then  $\emptyset$  is treated as a RawInputCharacter and  $\emptyset$  remains part of the escaped Unicode stream.*

### 2.3.3 Definite descriptions

Definite descriptions describe a referent. The description typically introduces new information on the referent that has not yet been given in the text (see [Mit02, 10]). Synonyms can be used in definite descriptions, as in the following example.

**Sample 2.5** *If the method is an instance method, it locks the monitor associated with the instance<sub>1</sub> for which it was invoked (that is, the object<sub>1</sub> that will be known as `this` during execution of the body of the method).* ([GJSB05, 554], emphases mine, monospacing in original)

The terms *instance* and *object* are synonymous in object-oriented programming, thus *the object* can be used to refer to its antecedent *the instance*<sup>5</sup>.

Besides synonyms, hyponyms and hyperonyms can be used in anaphoric definite descriptions – i.e. sub- or super-ordinate terms. It shall also be noted that definite descriptions can be more complex i.e. can involve quantities and attributes as in e.g. *the five green objects*. [Hen08] and [Sta09] included these features in their implementations but I will not do so.

The remainder of this chapter is mainly based on the work on Monika Schwarz-Friesel: [Sch00]<sup>6</sup>. While [Mit02], [Cla75] and [HH76] were also considered, Schwarz-Friesel's work was more valuable due to its cognitive and process-oriented perspective and its complex analysis of IA<sup>7</sup>. The next section will lay the cognitive groundwork for the introduction to indirect anaphora in the subsequent section.

## 2.4 Cognitive Foundations

Cognitive Science is an interdisciplinary field researching the human mind. Its subfield Cognitive Linguistics deals with models of language processing in the brain (among other things). Schwarz-Friesel explains how humans use their knowledge to process anaphora. Her explanations are based on models of how knowledge is structured in the mind and how it is activated so it can be accessed efficiently. These models will be summarized in the coming subsections.

**Future Work 2.2 (Include direct anaphora, based on Schwarz-Friesel's model)** *Schwarz-Friesel's model explains both the understanding of direct and indirect anaphora. As part of this work I will ignore the parts on direct anaphora and only use the parts on indirect anaphora. I did, however, become clear at a later stage of my work that it is necessary to implement direct anaphora along with indirect anaphora.*

<sup>5</sup>The antecedent *the instance* is actually referring itself, as is signalled by the definite article. It might be regarded an indirect anaphor anchored in the indefinite noun phrase *an instance method*.

<sup>6</sup>Some aspects of [Sch00] are summarized in [SF07].

<sup>7</sup>There are also works from computational linguistics that deal with indirect anaphora (see [PMMH04], [FBP05]). Their work is, however, based on using annotated text corpora or the web to gather the semantic and conceptual information required to resolve indirect anaphora. At the current stage of my work, this renders these works irrelevant, because the source code provides normative semantic and conceptual information.

### 2.4.1 Mental Representations

So called *modular* theories assert that the *mental lexicon* contains *semantic* entries and is separated from common sense or encyclopedic knowledge maintained by another mental module as part of *conceptual schemata*, even though both are connected and interact [Sch00, 32], even overlap [Sch00, 33]. Modular theories propose that the mental lexicon appears as a network in long-term memory (LTM) that connects words via semantic relations (e.g. synonymy, hyperonymy, and meronymy that will be explained later on); the entries of the lexicon shall describe the core meaning of a word [Sch00, 31]. The lexical meaning of a word is underspecified and independent from context [Sch00, 38], but language-specific (see [Sch00, 32]).

Conceptual schemata are described as complex knowledge structures in long-term memory that describe a typical instance of a subject or process; they are made up of *concepts* that are contained in schemata as variables called *defaults* that can be assigned a specific value in the process of comprehension or trigger *cognitive strategies* if the situation encountered does not fit the conceptual schema [Sch00, 34]. Conceptual schemata can be described as language-independent [Sch00, 32] and they are context-dependent i.e. parts of their contents may be relevant in some situations, but irrelevant in others [Sch00, 38].

Schwarz-Friesel briefly outlines two forms of conceptual schemata: frames and scripts [Sch00, 34f.]. While she highlights that frames detail typical components of objects of a certain class, Stillings et al. give a definition from artificial intelligence that encompasses all kinds of attributes, not only components: "A *frame* is a collection of slots and slot *fillers* that describe a stereotypical item. A frame has slots to capture different aspects of what is being represented. The filler that goes into a slot can be an actual value, a default value, an attached procedure, or even another frame (that is, the name of or a pointer to another frame)." ([SWC<sup>+</sup>95, 159], emphasis in original). A script, Stillings et al. write, "is an elaborate causal chain about a stereotypical event. It can be thought of as a kind of frame where the slots represent ingredient events that are typically in a particular sequence." ([SWC<sup>+</sup>95, 161], emphasis in original). Schwarz-Friesel mentions that scripts are augmented with properties, as well as pre- and post-conditions [Sch00, 35]<sup>8</sup>.

**Future Work 2.3 (Depth of conceptual decomposition)** *Schwarz-Friesel hints at the fact that it is not clear, how far conceptual decomposition goes [Sch00, 35]. A similar problem exists in computer science, where fine-grained decomposition increases reuse at the cost of complex dependencies. Computer science might find interesting insights from cognition research on this topic.*

For each lexicon entry there is a conceptual schema whose defaults act as the lexicon entry's *conceptual scope*. The lexicon entry and its conceptual scope form a so-called *cognitive domain* (see [Sch00, 38]).

---

<sup>8</sup>It is no coincidence that frames and scripts resemble similar concepts in computer science. It shows instead the influence of computer science that is part of the interdisciplinary field of cognitive science and thus influences the models made up in the field.

### 2.4.2 Text-world models

Having an idea of the structure of knowledge in memory, the process of text comprehension becomes of interest. Constructive theories of understanding assert that a model is constructed by the reader while receiving a text. A TWM is used to explain why it is possible to understand fictional or abstract issues as well as to talk about real-world objects that ceased to exist: the reader adds them to her TWM even if they do not exist *for real*. Schwarz-Friesel calls such a model *text-world model* (TWM) and describes that it contains nodes that are conceptual representations of the objects mentioned in the text from which the model was constructed [Sch00, 41]. To describe the construction of the nodes of TWM, three-tier semantics are best suited [Sch08, 189]. Three-tier semantics comprise abstract concepts, language-specific lexical meanings and current meanings determined by context [Sch08, 63]. The current meanings are represented by the nodes of the TWM. These nodes are not mere copies of lexicon entries but have been adapted based on existing information in the TWM and conceptual schemata [Sch08, 189]. As part of a process called *referentialization* the reader resolves the references in the text i.e. new nodes will be created in the TWM that act as referents, or existing nodes will be selected to serve as referents and this process includes elaboration of the TWM using knowledge from the reader's memory by means of cognitive strategies (see [Sch00, 45]). Because nodes in the TWM are derived from entries in the mental lexicon they have a cognitive domain as well.

**Future Work 2.4 (Instantiation of concepts and specification)** *The instantiation of nodes in the TWM described above and the off-line specification of nodes – i.e. turning a node into a more specific concept when further information concerning the referent is read – is described in the literature and will become more important for more complex uses of anaphora. [Sch08, 64f., 189f.] might be good starting points; it might actually be a good idea to read the entire book (for me too, I found it quite late during my thesis work when my reading time was up already).*

### 2.4.3 Focus and Activity

During referentialization, mental processes work on the contents of short-term memory (STM). Since semantic and conceptual knowledge is stored in LTM, knowledge must be chosen and transferred from LTM to STM. A selection is necessary because of the limited capacity of the STM and is based on processes managing focus and activity, of which the following ones can be distinguished (see [Sch00, 46], [Sch00, 137ff.] and [Sch08, 199]).

**Gaining focus** When a phrase is read, its node in the TWM is activated or re-activated (see below) and gains focus, i.e. is at the center of attention in STM.

**Losing focus** When the next phrase is read, the node of the previous phrase loses focus and the node of the new phrase gains it. The node of the previous phrase remains active in STM.

**Activation** The node of a phrase that is activated is also added to the TWM. Indefinite noun phrases, proper names, combinations of both and pronouns cause activation (see [Sch00, 70f.]).

**Semi-Activation** All nodes that are elements of a certain cognitive domain are semi-activated in LTM when one of the nodes that is an element of the cognitive domain is activated.

**Re-Activation** A definite noun phrase causes re-activation. That means that (a) its node has been inactive in LTM and becomes active in STM, and/or (b) the node of the phrase refers to an element of a semi-active conceptual schema in LTM that will in turn be activated in STM.

**De-Activation** A node in LTM becomes inactive when the node that caused its latest (re-)activation or semi-activation is removed from STM. This typically happens two sentences after the phrase corresponding to the phrase referring to the node had been read. The node may be re-activated later.

Now that mental representations and activation have been introduced in an abstract fashion, they will be exemplified during the discussion of forms of indirect anaphora.

## 2.5 Indirect Anaphora

In contrast to direct anaphora, the initial item involved in indirect anaphora is not called antecedent but *anchor*. Indirect anaphora typically has the following features [Sch00, 50].

**Anchor instead of antecedent** The previous text does not contain an explicit *antecedent*. Instead, an *anchor* is present, that is essential for the interpretation of the indirect anaphor.

**Conceptual relation** Anchor and indirect anaphor are conceptually close instead of being in a coreference relation that is typical for direct anaphora.

**Constructive interpretation** Interpretation of indirect anaphora includes constructive inclusion of conceptual knowledge instead of searching for the anchor only.

**No demonstratives or pronouns** Demonstratives and pronouns can only in rare cases be used as indirect anaphors.

According to Schwarz-Friesel, indirect anaphora can be seen as a form of referential underspecification that is driven by the writer's anticipation of the reader's knowledge that he assumes will be used by the reader to elaborate the TWM to overcome the underspecification [Sch00, 81]. Referential underspecification is to be distinguished from referential ambiguity in that in both cases the text lacks information required to establish a reference but only the latter leads to ambiguity because even context, semantic and common sense knowledge do not allow a single most likely referent to be identified [Sch00, 82]. Underspecification may hinder referentialization, if the reader does not possess the knowledge anticipated by the writer. It is, however, frequently<sup>9</sup> used because language is used economically [Sch00, 83].

<sup>9</sup>Schwarz-Friesel actually quotes studies based on corpora of Swedish and English texts that revealed that up to 60 % of definite noun phrases have no explicit antecedent (see [Sch00, 79]). It would be interesting to know if the same results can be found with corpora containing technical texts only. Looking for samples of indirect anaphora within the Java language specification gave me the idea that a lot of phrases were fully specified. This evidence is anecdotal only. It would be worth a detailed examination, though: Schwarz-Friesel reports on an experiment of her in which 40 % of the subjects found full specifications superfluous [Sch00, 79f.].

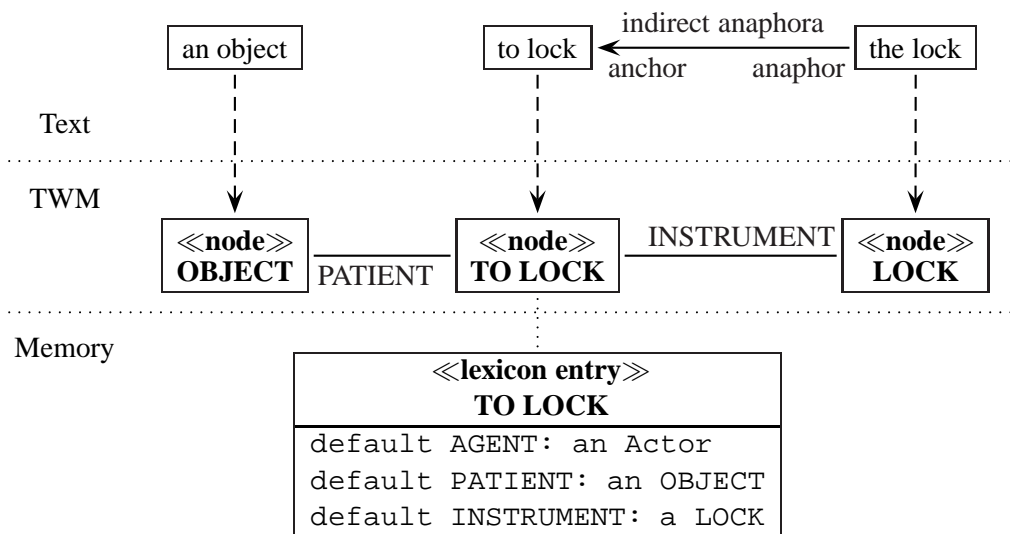


Figure 2.2: Relations in text and text-world model for sample 2.6

Schwarz-Friesel provides a classification of indirect anaphora based on the anchoring process used i.e. based on the process used to establish the relation between indirect anaphor and anchor. I will summarize this classification in the following sections.

### 2.5.1 Anchoring based on thematic roles

Indirect anaphora can be based on thematic roles (see [Sch00, 99ff.]). Thematic roles are used to classify the semantics of the arguments of a verb [Sch00, 100], the latter of which are identified as part of syntactic analysis. In the case of this kind of indirect anaphora, the indirect anaphor (IA) fills a thematic role (here: PATIENT or INSTRUMENT) of a previously mentioned anchor ( $\Downarrow$ ), as can be seen in the example below.

**Sample 2.6** *If the method  $m$  is synchronized, then an object<sub>PATIENT</sub> must be locked <sub>$\Downarrow$</sub>  before the transfer of control. No further progress can be made until the current thread can obtain the lock<sub>INSTRUMENT,IA</sub>. ([GJSB05, 478], markup mine)*

Figure 2.2 illustrates the relations in the text and text-world model of the sample that is now discussed. The figure contains the three phrases relevant for the interpretation of the indirect anaphor *the lock*, their nodes in the TWM as well as the lexicon entry related to the node of the verb phrase *to lock*. Note that this time reference relations are not named but simply shown as dashed arrows. In the second sentence of the text sample, the definite noun phrase *the lock* acts as indirect anaphor because it is marked with the definite article *the* signalling that it is known

## 2 Reference in Natural Languages

although it has not been mentioned before<sup>10</sup>. The other definite noun phrases of the sample are no examples of indirect anaphors based on verb semantics only<sup>11</sup>. In the case of *to lock* as used in the first sentence, three thematic roles can be identified according to the classification used by Saeed [Sae03, 149f.]: AGENT, PATIENT, INSTRUMENT (who locks something, what is locked, the lock used). The AGENT role is not specified in the text, but a default is contained in the lexicon entry. The phrase *an object* takes the PATIENT role – it is affected by the locking and moreover modified: after the locking it will be locked. The INSTRUMENT role is taken by *the lock* in the second sentence because it fits the role well: locks are used to lock things. The second sentence contains another verb (*obtain*) that has two thematic roles that are both taken by phrases of the second sentence even though no anaphora occurs related to this verb<sup>12,13</sup>.

Not in all cases is the thematic role taken by an indirect anaphor as specific as in the example just discussed. Consider another sample.

**Sample 2.7** *Otherwise, the value  $I_{PATIENT(1)}$  is added to the value of the variable  $PATIENT(2)$  and the  $SUM_{PATIENT(3),IA}$  is stored back into the variable.* ([GJSB05, 486], markup mine)

The indirect anaphor *the sum* takes the 3rd PATIENT<sup>14</sup> role of the verb *add* in the first sentence. This case is different from the first one, in that not only the verb's semantic entry in the mental lexicon and the indirect anaphor are involved in the resolution of the indirect anaphora. *To add* has a number of meanings: one may add a tree to a garden, one may add a final remark in a discussion to have the last word or one may add numbers during a calculation. The last meaning is used in the give example, but that meaning of *to add* needs to be invoked first to make the sentence sound<sup>15</sup>. The 1st and 2nd PATIENT roles *the value 1* and *the value of the variable* together with *is added* invoke a conceptual schema ARITHMETIC ADDITION that has defaults for the two given addends and a sum. The default for SUM is replaced by *the sum*, when the reader proceeded up to its mention in the text. The anchor in this example is thus found due to the conceptual scope of its lexicon entry. I.e. indirect anaphora based on thematic roles may not only involve semantic knowledge but also conceptual knowledge.

<sup>10</sup>Had it been introduced via the indefinite noun phrase *a lock* before, *the lock* would be a direct anaphor because it would refer to the antecedent *a lock*.

<sup>11</sup>(1) *the method m* is a direct anaphor that refers to an indefinite noun phrase of the previous sentence "A method *m* in some class *S* has been identified as the one to be invoked." [GJSB05, 477]. (2) *the transfer of control* is a rather direct anaphor that refers to the previous sentence "If the method *m* is not synchronized, control is transferred to the body of the method *m* to be invoked." [GJSB05, 478]. (3) *the current thread* is an indirect anaphor that can be resolved using inference only (see below) since the term is not introduced formally.

<sup>12</sup>The AGENT role is taken by *the current thread* and *the lock* takes the THEME role of *obtain* besides the INSTRUMENT role of *lock* that it already has. The THEME role is taken by entities that are affected but not modified by the corresponding verb.

<sup>13</sup>This example also shows the cohesive force of indirect anaphora: the verb-semantic relationship between *to lock* and *the lock* spans the two sentences, turning them into a coherent chunk of text.

<sup>14</sup>It becomes obvious here that generic thematic-role models have their limitations: it is hard to classify roles involved in abstract processes. It is also interesting to note that at least Saeed does not list a role for the outcome of an action that could be used for creative processes or calculations.

<sup>15</sup>\**Otherwise, the pine tree is added to the garden and the sum ...* would have been invalid because sums have nothing to do with gardening.

### 2.5.2 Meronymy-based anchoring

Not only thematic roles of verbs are modeled as part of the mental lexicon, the lexicon is also contains information about the relations between nouns. Hyperonymy has already been described as a nominal-semantic relation that can be used as the basis for direct anaphora on page 9. While in the case of hyperonymy identity of reference leads to the categorization of the anaphora as direct anaphora, identity of reference is not given for another nominal-semantic relation: meronymy [Sch00, 104ff.]. Meronymy is the name used for part-whole- and similar relations, as in the following example.

**Sample 2.8** *An if-then statement*  $\Downarrow_{(1)}$  *is executed by first evaluating* *the Expression*  $\Downarrow_{IA(1), \Downarrow_{(2)}}$ . *If the result*  $IA(2)$  *is of type Boolean, it is subject to unboxing conversion (§5.1.8). ([GJSB05, 372], markup mine)*

The text prior to the extracted sample contained a syntax definition that made clear that an if-then statement consists, among other parts, of an expression<sup>16</sup>. It is also known to the reader that an expression is evaluated, yielding a value that is the result of the evaluation and is, like all other values in Java, typed. A reader encountering the given sample can see from its indefinite article, that *an if-then statement* is a new entity in the text. The lexicon entry contains the parts of the if-then statement, among them an expression. The definite article of *the Expression* in turn signals the accessibility of the referred item to the user. No expression has been introduced before, but the currently focused node in the TWM is the lexicon entry of *an if-then statement*, *the Expression* can thus be understood as taking the role of the expression mentioned in the lexicon entry of *if-then statement*, i.e. *the Expression* is understood as indirect anaphor anchored in *an if-then statement*. The anchoring is triggered by the fitness of the IA as the part of the anchor. Similarly, *the Expression* acts as the anchor of the indirect anaphor *the result* in the following sentence.

It shall be noted that it would also be possible to eliminate this anaphora by rewriting the sentence "An if-then statement is executed by first evaluating its Expression." making the part-of relationship explicit in the text instead of deriving it from the lexicon entry<sup>17</sup>.

Schwarz-Friesel [Sch00, 108f.] differentiates types of meronymy and gives an example that shows that meronymy is often intransitive. She distinguishes relations between an object and its constitutive parts, an object and its materials, an object and portions of it, sets and their sub-sets and others but points out that loose association does not trigger meronymic anchoring.

The above example is a case of intransitive meronymy: removing *the Expression* to apply underspecification makes it hard to understand the connection between *an if-then statement* and *the result* because the indirect anaphora cannot be established and the two sentences do not form a coherent whole.

It would, however, be possible to underspecify in the context of a more detailed model whose meronymy relations would be understood as transitive due to the given lexicon entry for if-then statements. Consider the representation of an if-then-statement in an abstract syntax tree where

<sup>16</sup>The original text put expression in upper case and italics to highlight that the word refers to the preceding grammar definition

<sup>17</sup>This form is actually used frequently in the Java language specification.

the full specification of the relation between *the if-then statement* and *the expression would be*: "the expression (node) that is part of the children of the if-then statement (node)". It would be possible to shorten that sentence to "the expression of the if-then statement" without making the phrase less intelligible<sup>18</sup>.

### 2.5.3 Schema-based anchoring

There can be cases when the entry in the mental lexicon that an anchor refers to is not sufficient to establish a coherent relation between the indirect anaphor and the anchor. If the conceptual schema that acts as conceptual scope of the lexicon entry can establish such a relation, this is a case of *schema-based anchoring* (see [Sch00, 111ff.]). The following sample will be used to illustrate this kind of indirect anaphora.

**Sample 2.9** *It is instructive to consider what might happen without the verification step<sub>↓</sub>: the program might run and print:*

*s*

*This demonstrates that without the verifier<sub>IA</sub> the type system could be defeated by linking inconsistent binary files ([GJSB05, 342], markup mine)*

The resolution of the indirect anaphor *the verifier* can be explained as follows. The definite noun phrase *the verification step* semi-activates the script CLASS VERIFICATION that involves a default for a verifier that is replaced by *the verifier*, establishing indirect anaphora when the anaphor is read. Note that *the verifier* is not a part of *the verification step* and is therefore not part of the latter's semantics that might be stored in the mental lexicon<sup>19</sup>.

The definite noun phrase *the type system* seems to be less connected to CLASS VERIFICATION. It seems to be a good example for the final, most complex kind of indirect anaphora in Schwarz-Friesel's classification: inference-based anaphora.

### 2.5.4 Inference-based anchoring

Schwarz-Friesel defines *inference* as a process that activates conceptual knowledge from LTM or constructs mental representations required for the TWM; it thereby exceeds what can be done solely based on semantic knowledge from the mental lexicon [Sch00, 89]. Inference of indirect anaphora is limited by three factors: the semantic representation of the indirect anaphor, the anchor and the existing TWM [Sch00, 90].

Hence, inference-based indirect anaphora [Sch00, 114ff.] are those kinds of anaphora for which not only a default in a conceptual schema needs to be replaced in order to establish them, but conceptual knowledge from LTM must be activated or new nodes need to be created in the TWM to produce a coherent TWM of the text. Consider the following example.

**Sample 2.10** *Some thread<sub>ACTOR</sub> invokes<sub>↓</sub> the exit method of class Runtime or class System<sub>THEME</sub> and the exit operation<sub>DA</sub> is not forbidden by the security manager<sub>IA</sub>.* ([GJSB05, 331], markup mine)

<sup>18</sup>The question of how a compiler would process such a phrase is not to be discussed here.

<sup>19</sup>It was actually hard to find an example for schema-based anchoring for most potential samples involved meronymy, verb-semantic roles or inference.

The sentence describes one of the two cases that can cause a Java program to exit. While the text links *the exit method* to the classes that provide it, *the security manager* has a definite article as well, but was not introduced before. The indirect anaphor *the security manager* may be anchored in the METHOD INVOCATION script which is the conceptual scope of *invoke* and its ACTOR and THEME roles given in the text. In addition, the verb phrase *is not forbidden* hints users who possess that knowledge at the fact that METHODS CAN QUERY THE SECURITY MANAGER UPON INVOCATION TO MAKE SURE THE INVOCATION IS VALID that can be integrated into the TWM to establish indirect anaphora between the anchor METHOD INVOCATION and the anaphor *the security manager*.

### 2.5.5 Anchoring of Indirect Anaphors

The preceding discussion of samples described the referentialization of indirect anaphors, i.e. how a referent of an indirect anaphor is found by establishing a relation between the indirect anaphor and its anchor in the TWM. It was however not detailed how a suitable anchor is selected in the presence of multiple potential anchors. The following two steps are given in [Sch00, 135] to explain the selection of a suitable anchor from a set of potential anchors. The steps are called *cognitive strategies* for they happen automatically and unconsciously.

1. Identify a suitable anchor within the text, i.e. a textual item (the anchor) whose cognitive domain has a placeholder for the referent of the indirect anaphor. The search for a suitable anchor requires that anchor and indirect anaphor are reasonably close to each other within the text. A suitable anchor is typically<sup>20</sup> determined by the following conditions [Sch00, 139ff.].
  - a) Referential unambiguity: there may be more than one *potential* anchor for an indirect anaphor but only one anchor can have a *suitable* referent for the indirect anaphor in its cognitive domain.
  - b) Plausibility: the anchoring must be plausible on-line i.e. during comprehension of the indirect anaphor. This is the case, when a role in the cognitive domain can be set by the indirect anaphor ad hoc or using inference (see step 2. below that will *actually* perform what is only required to be *possible* by this condition).
  - c) Anchor theme is focused: When the indirect anaphor is processed on-line, the theme of the potential anchor, which can be its conceptual scope, must be focused in order to allow for a relation between anchor and indirect anaphor to be established easily. If the theme of the anchor is not currently focused because e.g. there is another sentence with a different theme between anchor and indirect anaphor, the relation will be hard to establish. Note future work item 2.5 below.
2. Establish the relation between indirect anaphor and the suitable anchor by either
  - a) searching the cognitive domain of the potential anchor for a role that the indirect anaphor can take<sup>21</sup>, or

---

<sup>20</sup>i.e. the conditions may be incomplete or overly restrictive in exceptional cases

<sup>21</sup>The role can be a thematic role in the lexicon entry of a verb or a meronymic role in the lexicon entry of a noun

## 2 Reference in Natural Languages

- b) searching the cognitive domain of the potential anchor for a default in the conceptual scope of the lexicon entry that the indirect anaphor can replace, or
- c) adding a role that can be replaced by the indirect anaphor via inference.

Schwarz-Friesel expresses that she takes a moderate minimalist position concerning when to use text-semantic and when to use conceptual knowledge to resolve anaphors and therefore supposes that referentialization is efficient and non-redundant a process and hence later options will only be taken when no prior option allowed to establish a reference [Sch00, 22] (e.g. 2c will only be applied when 2a and 2b did not yield a suitable anchor).

**Future Work 2.5 (Details on thematic progression)** *In order to be able to detect changes in the theme of a text between one sentence and another, it is necessary to have a good idea of the representation of a sentence's theme. Schwarz-Friesel discusses thematic progression in the case of indirect anaphora [Sch00, 97]: she argues that rather a "scalar" than the existing "binary" theory of information is necessary, that the latter classifies the topic of a sentence as either given (a theme) or new (a rheme), even though indirect anaphors do continue the theme of the anchor as well as introducing new referents (that function as rhemes upon introduction, but from later points in the text are regarded as themes). Schwarz-Friesel asserts that there may be multiple themes that are gradually activated. Besides this potential multiplicity of themes, Schwarz-Friesel sees themes as represented by conceptual schemata [Sch00, 142] but also points out that it is unclear how fine-grained conceptual schemata are [Sch00, 35]. To me, what follows from this is that the granularity of conceptual schemata determines whether or not a switch of theme exists. In the extreme case of an all-encompassing scheme, a switch of theme can never occur, in the other extreme of totally isolated minimal schemes, every new word would bring a switch of theme. All this shows that the idea of a theme needs further clarification in order to effectively determine the scope searched for potential anchors of an indirect anaphor.*

## 2.6 Summary

In this chapter reference and proper names have been defined. Anaphors have been introduced that relate to an item backwards in the text to draw parts of their meaning from. Anaphors maintain the topic of a text or introduce new information and abbreviate through the use of lexically shorter forms or underspecification. A gradual model spanning from direct to indirect anaphora was mentioned in which direct anaphors relate to an antecedent given in the pre-text and often co-referential while indirect anaphors relate to an anchor in the pre-text whose referent provides a default that the referent of the indirect anaphor can set. A modular approach to knowledge representation was introduced, that separates lexical from conceptual knowledge. A model of three-tier semantics has been pointed out that describes how a reader receives a text, creates a text-world model from it and elaborates the text-world model based on the knowledge representations determined by focus and activity. Examples of anaphora from the Java language specification have been analyzed and an overview over the cognitive strategies involved in anchoring indirect anaphors was given. Areas that need further attention have been pointed out.

# 3 The Relations Between Natural Languages and Programming Languages

Now that it is clear what indirect anaphora is, to prepare its transfer to the Java programming language, the relations between natural languages and programming languages are considered.

## 3.1 Programming Languages considered Languages

Three ways of relating natural languages and programming languages to each other can be identified ad hoc: (1) fragments of natural language are contained in source code of programming languages as identifiers, (2) both are sub-concepts of the same superordinate concept of *language* and (3) they might be metaphorically connected. Relation (1) will be discussed in section 4.1, (2) will not be disputed – it holds e.g. for Chomsky's syntax-oriented definition of language as "a set (finite or infinite) of sentences, each finite in length and constructed out of a finite set of elements." [Cho57, 13]. Relation (3) will be elaborated in the following.

Through the course of the history of programming languages it has been remarked that there are analogies between natural languages and programming languages (see [Zem66, 141] and [Nau92, 26]). Carsten Busch treated the analogies between natural languages and programming languages as metaphorical [Bus98, 164f.]: according to him, the word *language* that is part of *programming language* had been transferred from the context of *natural languages* to the context of what used to be called a *coding system* before; he asserts that through this transfer, the new concept of a *programming language* was created and shaped. I am not yet convinced of this hypothesis because it is not clear to me which concepts out of the context of natural language had, during the 1950'ies not been available as part of the context of language in general thereby making a metaphorical transfer necessary from the context of natural languages. Regardless of whether or not this can be found for the past, it provides an option for the future and I deem transferring indirect anaphora from natural languages to programming languages a way to elaborate this metaphorical relation because indirect anaphora cannot be copied from English to Java due to the grave structural differences. The transfer will thus include mapping (not necessarily syntactic) structures of natural language to structures of Java.

## 3.2 Naturalistic Programming Languages

If the notion of (*natural*) *language* being used as a metaphor for a coding system is taken to the extreme, the coding system will look entirely like natural language. This can be seen as the

### 3 The Relations Between Natural Languages and Programming Languages

goal of natural-language programming i.e. programming in natural language. A more recent approach reaches closer: *naturalistic programming languages*. Breaking up the term *naturalistic programming language* into its constituent parts, it unfolds into a programming language that is naturalistic i.e. "derived from or closely imitating real life or nature" [Dic11a]. Instead of departing from natural language, as in *natural-language programming*, this term departs from programming languages and qualifies them as imitating real life, respectively its languages: natural languages. Lopes et al., who first used the term *naturalistic programming*, remind of the fact that in programming languages "it should be possible to construct abstractions on top of a relatively small number of primitive abstractions" and that "such primitive abstractions should be inferred from wider ground of Linguistics" [LDLL03, 203]. They propose that such abstractions are the binding mechanisms used in natural language and that naturalistic programming languages "take their direction from the structure and expressiveness of natural languages rather than from the idealized models of traditional programming languages." [LDLL03, 203].

A note on the adjective *natural* that occurs quite often in writings on naturalistic programming when it comes to justifying language design decisions: Jef Raskin mentioned in his book "The Humane Interface", that the adjectives *natural* and *intuitive*, when used to describe user interfaces, mean that something is known or easy to learn [Ras00, 150-1]. It might be true that a particular feature taken from a natural language is known or easy to learn. However, for its implementation in a programming language to be called natural, known, or easy to learn, it should at least be outlined why the implementation of the feature matches its occurrence in natural language. This is necessary because not all implementations match the feature of natural language they seek to implement, which is a frequent problem in systems made for natural language programming.

**Future Work 3.1 (Criticizing the concept of *natural language*)** *The Oxford Dictionary defines the adjective natural as meaning "existing in or derived from nature; not made or caused by humankind" [Dic11b]. It could be asked to what extent languages used by humans are given by nature respectively are shaped by humans themselves. Neurolinguists will provide evidence showing that natural languages are shaped by our biology, but there will be as much evidence for the idea that natural language is an abstract construct of humans trying to understand it. Criticizing the concept of natural language would mean to elaborate it, trying to gauge whether it is more given or more made. I find it very likely that such criticism has been written already. It could provide feedback for attempts to evaluate the naturalness of naturalistic programming. The most extreme outcome of the criticism could be to use humane programming instead of naturalistic programming – analogous to Raskin's term humane interface.*

### 3.3 Summary

This chapter briefly outlined the relations between natural languages and programming languages: that (1) fragments of natural language are contained programming languages, (2) both are sub-concepts of the same superordinate concept of *language* and (3) they might be metaphorically connected. The metaphorical relation I consider hypothetical for the past but will pursue it in my own transfer. Naturalistic Programming was shown to be along the lines of that metaphorical relation and criticism of usage of the term *natural* was added.

## 4 Reference in Java

Now that forms of reference in natural language and the relationship between natural language and programming languages have been analyzed, reference in Java (see [GJSB05]) will be looked at in order to see whether or not forms of anaphora are already possible in the Java. I will therefore use words from chapter 2 as metaphors to put them in place of words used to describe reference in Java and then analyze whether the metaphors and their Java-related context fit the requirements that have been stated for the different kinds of reference in chapter 2.

Similar to the focus put forward in chapter 2, I will only regard local means of reference. That means that in this chapter only forms of reference will be considered that occur within the bodies of methods, constructors and initializers in Java. I will ignore references that can not typically be used without crossing the boundary of a file of source code, e.g. inheritance, interface implementation, use of types and package names. Analogous to the means of reference in natural languages treated in sections 2.1, 2.2 and 2.3, occurrences of names, deixis and zero anaphors in Java will be discussed in this chapter.

### 4.1 Names

Names in Java exemplify the first relation between natural language and programming languages from section 3.1: natural language appearing in names used in programming languages.

According to the Java language specification, names are in Java used to refer to declarations of e.g. classes, fields, methods and local variables and they can be either simple, consisting of a single identifier, or qualified, consisting of multiple identifiers separated by dots [GJSB05, 113]. An identifier cannot include spaces, but underscores ("\_"), dollar signs ("\$"), letters and digits [GJSB05, 19]. If no further assumptions about names are made, the names in a successfully compilable Java program are proper names as per van Langendonck's definition summarized in section 2.2: they refer to a unique entity (a declaration), that is thereby highlighted within the class of entities to which it belongs (e.g. field declarations) and the meaning of the name does not determine what the name refers to (which is true since Java has no notion of meaning that could be found in names<sup>1</sup>). Java compilers take this perspective with regard to names in the programs that they compile.

It is trivial, but programs are of course not read by compilers only, but also by their authors and other programmers. Programmers can take the perspective of the compiler and suppress their knowledge of natural language when reading programs. They are, however encouraged to give meaning to names used in Java programs and that meaning comes from natural language. The following naming conventions were taken from the Java language specification.

---

<sup>1</sup>This is the case because the compiler does not parse the contents of the character strings used as identifiers. Instead, the meaning of a name in Java is what the name refers to (see [GJSB05, 126ff.]).

## 4 Reference in Java

1. Names of class types should be descriptive nouns or noun phrases, not overly long, in mixed case with the first letter of each word capitalized. [GJSB05, 149]
2. Method names should be verbs or verb phrases [GJSB05, 149]
3. Fields should have names that are nouns, noun phrases, or abbreviations for nouns. [GJSB05, 150]
4. Local variable and parameter names should be short, yet meaningful. They are often short sequences of lowercase letters that are not words. [GJSB05, 151]

Nouns, noun phrases, verbs and verb phrases are distinguished in the syntax of natural languages as parts of sentences and have a meaning in natural language. Using them as names in Java gives them a second meaning that is only accessible to programmers, but not to compilers. This creates a gap in these names between the semantics of Java that both compiler and programmer know and the semantics of natural language that are only accessible to the programmer. This semantic gap makes it possible to write code whose contained names imply semantics that cannot be ensured by the compiler. That is a drawback of the use of descriptive names from natural language in Java (and other programming languages). Considering the fact that the natural-language meaning of a Java name is, from the perspective of the programmer, connected to what it refers to, one might want to categorize them as anaphoric because anaphors are semantically related to their referent. Since this meaning is only available to programmers but not to compilers, I am reluctant to categorizing Java names as anaphors, which might be a good application of Consten's gradual scale between deixis and anaphors (see 6). This objection does of course not apply to names that do not use full words as proposed in the naming convention for local variables and parameters.

### 4.2 Deixis

Besides names, a lot of programming languages have keywords that lexically resemble pronouns from natural language (see section 2.3.1). While names in Java can be used to refer to a variable that holds a reference to an object at runtime, there are forms of deixis in Java and other programming languages that hold a reference to an object directly. I categorize these forms as deixis because they are neither forms of direct anaphors, nor forms of indirect anaphors. The forms are no direct anaphors, for Java has no notion of gender and number in which anaphor and antecedent could agree and there is no antecedent in the text to which an anaphor relates and could be close to (see Schwarz-Friesel' canonical conditions mentioned in section 2.3). The forms are no indirect anaphors due to the lack of an anchor visible in the text and because these forms do not convey meaning that could invoke a cognitive domain in order to establish a relation to the cognitive domain of a potential anchor (see section 2.5).

Java and other programming languages have a keyword `this` that can be used within instance methods to refer to the corresponding instance and lexically resembles the demonstrative pronoun *this* known from English where it is used to refer anaphorically or deictically (i.e. within or out of the text). `this` is a pseudo-variables: it cannot be assigned a value. Instead, its value

is determined by the runtime system in accordance with the language specification. This is different from e.g. local variables that refer to explicitly defined antecedents (the declaration of the local variable) that can be freely positioned within e.g. a method body as long as the declaration precedes the first anaphor (the first use of the local variable) referring to it. Thus new forms of reference should be sought that have an antecedent or anchor in the text that can be freely positioned by the user of the PL as is the case with local variable declarations except for the fact that the antecedent or anchor is not explicitly named in the text<sup>2</sup>.

### 4.3 Zero anaphors

Besides names and deixis, zero-anaphors can be found in Java: in a certain kind of method invocation. Method invocation expressions typically specify the object upon which the method is to be invoked. E.g. in `this.foo()` the primary expression `this` is evaluated at runtime to retrieve the object upon which the method `foo()` is going to be invoked. It is common to shorten this expression to `foo()` instead of `this.foo()`. That will lead the compiler to search for a matching method declaration in an enclosing type declaration that is both visible and accessible [GJSB05, 400]. Like in the case of natural language (see section 2.3.2), use of zero-anaphora is syntactically restricted: it can only occur in the initial position of a complex expression (e.g. in front of but not after `foo()` in `foo().bar()`). The fact that the members of the enclosing type are searched is similar to how schema-based indirect anaphora is resolved (see section 2.5.3). However, like in the case of the pseudo-variables mentioned above, the antecedent is determined by the language specification and cannot be positioned in the method body but only be a member of an enclosing type.

### 4.4 Requirements for Indirect Anaphora in Java

Existing means of reference have been considered, and none was clearly identified as indirect anaphora. What then could make an indirect anaphor in Java? Schwarz-Friesel identified a number of features of indirect anaphora (see [Sch00, 118]), a number of these features I deem appropriate for indirect anaphora in Java as well:

**domain-binding** All forms of indirect anaphors are domain-bound: what they refer to is determined by one or more cognitive domains.

**strategies constitute referents** The reference that connects a definite noun phrase functioning as indirect anaphor to its referent is established by cognitive strategies in the text-world model. These cognitive strategies set placeholders from the cognitive domain that have been integrated into the TWM.

**implicit coherence relation** The cognitive strategies do not only constitute a referent, but in the course of that also establish an implicit coherence relation between the indirect

---

<sup>2</sup> I did not find a requirement of free positioning in definitions of anaphora in natural language but that does not irritate since it is programming languages that introduced keywords that can be used as anaphors in natural language but fixed the position of the antecedent.

#### 4 Reference in Java

anaphor and its anchor. This relation is expressed by the relation within the TWM between the referents of indirect anaphor and anchor.

**rhematic thematization** Indirect anaphors continue given (thematic) information while introducing new (rhematic) information at the same time.

Two further aspects follow from the prior contents of this chapter.

**careful lexical resemblance** Keywords used for indirect anaphors in programming languages should not resemble words from natural language if their semantics are only partially equivalent.

**freely positioned anchor** Programmers should be able to freely position anchors within areas of the source code. This implies that there is an explicit anchor in the text.

### 4.5 Summary

A comparison of means of reference in natural language and Java took place in this chapter. It was found that from the perspective of a Java compiler, names in Java satisfy van Langendonck's requirements for proper names as known in natural language. The naming conventions of Java do, however, propose that names in Java be phrases of natural language, making them anaphoric from the perspective of a programmer. This double function creates a semantic gap in each Java name if it resembles words from natural language. This resemblance can potentially confuse programmers. Moreover, it was found that Java supports deixis and zero anaphors. Before further forms of indirect anaphora are added to Java, Schwarz-Friesel's features of indirect anaphora have been given and extended by careful lexical resemblance of words from natural language and free positioning of anchors.

## 5 Indirect Anaphora for Java

Selected means of reference in natural language and the Java programming language have been introduced in chapters 2 and 4. It has been shown in chapter 3 that natural languages and programming languages can be regarded as being metaphorically related. In this chapter I will extend the metaphor of coding systems regarded as languages by introducing another metaphor to source code: indirect anaphors (see section 2.5). The metaphor *indirect anaphor* will stand for a new indirect means of reference in a modified version of Java that can be used within the bodies of methods, constructors and initializers. Based on the analyses from the previous chapters, this metaphor will be developed in an abstract fashion in the next section. The subsequent sections will detail specific forms of the metaphor.

### 5.1 Constructing a Metaphor

Busch did not only analyze the use of metaphors in programming and computing (see section 3.1) but reviewed definitions of metaphor as well. Based on his summary of informal definitions of metaphor (see [Bus98, 10ff.]), I will transfer the phrases *indirect anaphora*, *indirect anaphor* and *anchor* that are used within the context of natural language to the context of programming languages, specifically Java. The dialect of Java created by the transfer I will *Jaaa* to have a name to refer to it<sup>1</sup>. Busch underlined that not only the two contexts involved in the transfer are important for the metaphor, but that their *interaction* is equally relevant. Interaction means: to develop the meaning of the metaphor in the new context, elements from the original context must exist that can be mapped to elements in the target context [Bus98, 13ff.]. Developing a metaphor might be seen as the process of initially copying a word and its conceptual schema from one context to another and then rooting the copy of the conceptual schema in the new context. The rooting may happen by replacing the slots in the copied schema with slots that can be filled with elements from the new context<sup>2</sup>. To describe the interaction of the source and target context, I will provide a comparison of potential contents of the conceptual schemata of *indirect anaphora* in the contexts of natural language and programming language in the following<sup>3</sup>. Table 5.1 provides an overview of the two contexts compared that I hope is useful while reading this chapter.

---

<sup>1</sup>Only minimal effort was made to find this name.

<sup>2</sup>Busch provides a formalization of metaphor (see [Bus98, 59ff.]), but an informal definition is sufficient for this thesis.

<sup>3</sup>Upon more detailed inspection than performed here, cases may be identified that make the comparison become inappropriate. This will likely happen for the purpose of this chapter is not to *prove* a correct mapping between the two contexts but to create a plausible interpretation of the metaphor only.

## 5 Indirect Anaphora for Java

Element	Natural Language	Jaaa
<i>Pragmatics</i>		
Participants	Humans, rarely computers	Humans as writers, computers and humans as readers
Sovereign over definition	Technically: none	Compiler
Text	Coherent sentences	Body of a method, constructor or initializer
<i>Syntax</i>		
Top-level structure	Sentence	Statement
Phrases	NP; VP	Expression; method invocation expression
Pronouns	<i>this</i> and many others	<i>this</i> , <i>super</i>
Ellipsis	Supported	Supported
Indirect anaphor (here)	the <noun>	.Name
<i>Cognitive Foundations</i>		
Comprehension happens	While reading	At compile-time
Construct of comprehension	Text-world model	Abstract syntax tree
Kind of model	Schwarz-Friesel: modular	holistic
Lexicon entry	Entries in mental lexicon	Headers of: invocables, classes, interfaces
Conceptual schema	Frame; Script	Frame of a type: fields, accessors, direct and indirect supertypes; Script (potentially): body of an invocable
<i>Semantics</i>		
Thematic roles	AGENT, PATIENT, THEME ...	Argument- and return types of methods
Nominal relations	Hyperonymy, Meronymy ...	Supertypes, fields and accessors
Referent of IA	Lexicon entry in TWM	Header of: class or interface
Referent of Anchor	node in TWM	header of: method, class or interface

Table 5.1: Elements compared in order to establish the metaphor of indirect anaphora (IA) in the context of Jaaa, a modified version of Java that supports indirect anaphors

### 5.1.1 Pragmatics

Starting from an external perspective, it can be observed that while natural-language texts are typically written and read for and by humans, source code of programs is typically written by humans<sup>4</sup>, but read by humans and compilers. Further, in the latter case compilers have a sovereignty of definition over what counts as valid program text and what semantics a program has.

The notion of text used in my implementation will be a quite restricted one. As already stated in the introductory paragraph of this chapter, I will apply indirect anaphora within the bodies of methods, constructors and initializers only. Each such body I will consider as a separate text that is unrelated to all other bodies in a file of source code. This differs from the definition of text in linguistics that does not define such strict borders and defines texts via the coherence established by anaphora and other means.

### 5.1.2 Syntax

Before the meaning of a text can be analyzed, the syntax used by its language needs to be considered. The highest level syntactic structure in natural language is a sentence. I identify natural language sentences with Java statements. This is plausible because within a method body, statements are the highest-level structure in Java and because statements can have nested sub-statements just like sentences can have nested sub-sentences (called clauses).

The next smaller syntactic element after clauses are phrases of natural language. They have a head by which they are distinguished into noun-, verb-, prepositional and other phrases. I compare noun phrases (NPs) of natural language to expressions in Java because expressions are evaluated to a value and have a type. The Java language specification says that the result of an expression *denotes* a variable, a value or nothing, the latter being the case for invocations of methods that return void [GJSB05, 409]. I will use the term *denotation* for the variable or value that an expression *denotes*. The verb *to refer* I will use in the same way as I did in the preceding chapters: an item in the text can *refer* to a lexicon entry that has the role of a *referent* in the *reference* relation between the item in the text and the lexicon entry. Verb phrases (VPs) comprise a verb and optionally arguments to the verb. I compare VPs to method invocation expressions – because both express a concrete action. These two interpretations are covered by the naming conventions of the JLS (see section 4.1). This comparison is inconsistent from the perspective of natural language: VPs cannot be NPs, while in Java method invocation expressions are valid expressions because, like all other expressions, they are evaluated to a value at run-time<sup>5</sup>.

It has been shown in section 4.2 that pronouns and ellipses are both used in natural language and in Java.

A final syntactical question concerns the form of indirect anaphors. In chapter 2 I did only discuss indirect anaphors that have the form *the* <noun> wherein *the* is the definite determiner signalling that the referent is known to the reader and <noun> is a noun hinting at the ref-

---

<sup>4</sup>exceptions are subsumed under the term *generative programming* that describes the use of computer programs that generate source code

<sup>5</sup> Method invocation expressions for methods that return `void` are an exception to this.

## 5 Indirect Anaphora for Java

erent. Analogous to that I will only handle indirect anaphors that take the form `.Name` or `.Name ( Argumentsopt )` in Jaaa. `Name` is a simple of qualified Java name that must contain a reference to a type e.g. `java.lang.SecurityManager`, `SecurityManager` or `System.out`.

The `.` (dot) acts as definite determiner. The indirect anaphor is a primary expression (see [GJSB05, 420]) in Jaaa and can, due to the lack of recursion in its definition, occur on the leftmost end of a chain of binary expressions in which each binary expression connects two expressions.

I chose to prefix the type used in an indirect anaphor using a dot instead of using a yet to be introduced keyword `the` because I only implement a very small subset of what `the` is used for in natural language and I do not even support direct anaphors what would make the use of `the` even more irritating to users than the lacking implementation of direct anaphors. The choice of the dot does have a positive aspect as well: it is typically used to connect the parts of a chain of field accesses and method invocations. Now that it is used at the beginning, one could assume that there is something missing in front of the dot, which is true: the information missing in front of the dot will be derived from the anchor that will be identified and an error will be raised if no suitable anchor is available.

### 5.1.3 Cognitive Foundations

Compilation is usually regarded as a constructive process in the sense that it creates a compiled binary and the compiler creates an abstract syntax tree (AST) to elaborate the parsed text structure. Similarly, the description of the cognitive model of anaphora processing of Schwarz-Friesel referred to in chapter 2 includes the construction of a text-world model (TWM). This reliance on the AST is the reason that made me choose to implement statically-resolved indirect anaphora over resolution at runtime. Static resolution also corresponds to the fact that there are technical documents like specifications available in natural language that describe how to do things without the reader actually performing the actions, which is the equivalent of the division between compile-time and run-time in computing.

Concerning knowledge that contributes to the construction of the TWM there are two lines of theories in cognitive linguistics: holistic theories assume that all knowledge is contained in conceptual schemata, modular theories suppose instead that lexical and encyclopedic knowledge is stored separately in the mental lexicon and in conceptual schemata even though both interact and can be redundant. The model proposed by Schwarz-Friesel is a modular one, but Java does not have a lexicon<sup>6</sup>.

I treat method headers (see [GJSB05, 209f.]) as lexicon entries in Jaaa because parameter and return types in method headers resemble thematic roles of a verb's lexicon entry (even though types are more specific than thematic roles). The same is true for constructor declarations (see [GJSB05, 240]): their parameters resemble thematic roles and even though they do not return a value explicitly, constructor invocations result in a value of the type used as the class or interface type name in the class instance creation expression. The *constructor header* is the constructor

---

<sup>6</sup>At least not one exceeding the fixed keywords. It would however, be an option to develop systems for naturalistic programming that support multiple languages in a modular way that separates the lexicon from conceptual knowledge.

declaration without the constructor body. I will use the term *invocable* to refer to methods and constructors, *header of an invocable* to refer to the header of a method or constructor, *body of an invocable* to refer to the body of a method or constructor.

I treat class and interface headers<sup>7</sup> as if they were lexicon entries of nouns because the type graph in a Java program resembles a semantic network of hyponymy (subtype) and hyperonymy (supertype) relations that connect nominal lexicon entries in models of natural language<sup>8</sup>.

The fields and accessors<sup>9</sup> declared or inherited by a class or interface do, together with its direct and indirect supertypes, form what I will call the frame of the class or interface. This choice is obvious since the definition of Stillings et al. reproduced in section 2.4.1 is very close to what a class contains in natural language. I ignored their mention of procedures because I want to model indirect anaphora only, not execution of methods or procedures, which is already implemented in Java.

**Future Work 5.1 (Scripts in Java)** *It would also be possible to use Scripts to store encyclopedic knowledge (see 2.4.1). Methods seem to be similar to scripts, especially given Schwarz-Friesel's mention of pre- and post-conditions that reminds of design by contract. However, unlike a method which is associated to its declaring type and its subtypes scripts are not exclusively associated to an entity. Scripts are not investigated as a form on conceptual schema in this work, they might be considered in future work, though.*

### 5.1.4 Semantics

Now that the cognitive foundations have been laid, it is possible to describe semantic elements of the two contexts. Beginning with verb semantics, thematic roles of natural languages are reflected in the types of the declared parameters and return types of headers of invocables in Java. These declared types are, however, more specific than the generic thematic roles. Relevant for semantics of nouns are the relations between them. Natural language distinguishes is-a relations like synonymy and hyperonymy as well as part-of relations (meronymy). While is-a relations are manifest in Java source code in the form of *extends* and *implements* clauses that define subtype relations, meronymy is in Java expressed via fields and accessor methods of a class. This comparison is fuzzy, though. In the absence of further study I would guess that there normally are no synonymy relations in the type graphs used within Java programs because one typically strives to avoid redundancy in source code.

The semantics of an indirect anaphor can be defined as follows. An indirect anaphor is an expression that denotes another expression. The indirect anaphor is at runtime evaluated to a

---

<sup>7</sup>By class and interface header I mean the name of a class or interface and all its supertypes. This diverges from the definition of a nominal lexicon entry in the context of natural language used in section 2.4.1. In section 2.4.1 the lexicon entry contained mandatory parts of the noun, that I do not include. This choice is due to the fact that fields of classes do not necessarily contain parts of an object but can as well contain objects that are in some other way associated to the object or optional parts of the object that would in section 2.4.1 be part of the conceptual schema and hence will be part of the conceptual schema in Java as well.

<sup>8</sup>The other kinds of types known in Java – type parameters and array types – are not treated as part of this work.

<sup>9</sup>Accessor methods are used to get or set the value of a field of a class. I will define which methods I consider to be *beaccessors* in section 5.4.1 below. Other methods than accessors will be ignored here because they cannot be used at runtime to navigate the relations between the classes without risking side-effects or having to provide arguments.

result, which is the result of the denoted expression. If the result of the denoted expression is a value, the result of the indirect anaphor is a value. If the result of the denoted expression is a variable, the result of the indirect anaphor is a variable. The indirect anaphor refers to a referent, which is the header of the class or interface that is the type of the result of the denoted expression<sup>10</sup>. The result type of an indirect anaphor is the type declared as part of the indirect anaphor (I will also refer to this type as "the type of the indirect anaphor"). The type of the result of the denoted expression must be assignment compatible (see [GJSB05, 95]) to the result type of the indirect anaphor. This can be ensured at compile time since both the result type of the direct anaphor and the type of the variable or value denoted by the indirect anaphor are known at compile time. An anchored indirect anaphor stands in a relation of indirect anaphora to its anchor. An anchor is an expression that refers to a lexicon entry that is either (a) the header of a class or of an interface or (b) the header of an invocable<sup>11</sup>. The relation between indirect anaphor and anchor is manifest in the AST in a way specific to the type of indirect anaphor. Types of indirect anaphors and the forms that the relations to their anchors take in the AST are given in the following sections.

### Anchoring

Before the specific relations between anchor and indirect anaphor can be described, the cognitive strategies used to select a suitable anchor among a set of potential anchors from section 2.5.5 need to be transferred to Jaaa. The transfer results in the following strategies.

1. Identify a suitable anchor within the method body, i.e. an expression (the anchor) whose cognitive domain has a place for the referent of the indirect anaphor. A suitable anchor is determined by applying the following conditions<sup>12</sup> that filter the set of potential anchors. The set of potential anchors contains all expressions that occur within the body of the method, constructor or initializer containing the indirect anaphor before the indirect anaphor.
  - a) The anchor must occur in a statement before the statement that contains the indirect anaphor<sup>13</sup>.
  - b) No void: If the anchor is a method invocation expression, the method invoked must not return void.
  - c) No primitives: If the result of an expression is a primitive, it can be focused and activated but it cannot function as anchor because primitives are not made up from parts that could be useful in anchoring.

---

<sup>10</sup>An indirect anaphor can not yet denote an expression whose result has a primitive type. This design choice is motivated by the fact that primitive types have no header or frame.

<sup>11</sup>Furthermore, the lexicon entry referred to by the anchor expression has a conceptual scope which is a frame.

<sup>12</sup>The strategy in section 2.5.5 was limited to *typical* cases. Here, exceptions will not be allowed because a compiler is supposed to follow clear rules. If it is found that the rules do not meet the expectations of programmers they might need to be adapted.

<sup>13</sup>This is choice made to avoid cases like the right-hand operand of an assignment being an indirect anaphor that denotes the left-hand operand of the assignment. If it is later found that such constructs are not irritating, they might be implemented.

- d) No literals: Literal expressions cannot be used as anchors<sup>14</sup> .
  - e) No arguments of explicit constructor invocations: Explicit constructor invocations that can appear as the first statement in a constructor body can have arguments that cannot function as anchors<sup>15</sup> .
  - f) Referential unambiguity: there may be more than one *potential* anchor for an indirect anaphor but only one anchor can have a *suitable* referent for the indirect anaphor in its cognitive domain. Note that multiple occurrences of an expression in the source code can lead to referential ambiguity (see section 5.2.1 for a discussion) if the occurrences denote the same variable (TODO: correct wording?).
  - g) Plausibility: the anchoring must be plausible on-line i.e. during compilation of the indirect anaphor. This is the case, when a role in the cognitive domain can be set by the indirect anaphor ad hoc or using inference (see step 2. below that will *actually* perform what is only required to be *possible* by this condition).
  - h) Theme ignored: This is actually not a condition, but a reminder of the condition that was expressed in case of natural language: that the theme of a suitable anchor must be focused. As has been explained in future work item 2.5 I did not find a good model of theme. Furthermore, the conceptual schemata that Schwarz-Friesel used as themes cannot be used in the transferred version because they are oriented towards frames that are relatively fine-grained while Schwarz-Friesel used scripts that looked more promising but aren't available here. Jaaa will thus not use a themes right now. Focus will thus be irrelevant as well right now.
  - i) Last 2 statements: Instead of requiring that the theme of a suitable anchor be focused, it will be required that the headers of the methods, classes and interfaces referred to by suitable anchors must be active. This is declared to be the case if they have been read during the last two statement prior to the statement containing the indirect anaphor.
2. Establish the relation between indirect anaphor and the suitable anchor in one of the following ways.
- a) If the anchor refers to the header of a method and the return type of the method is assignment compatible (see [GJSB05, 95]) with the type of the indirect anaphor, or the anchor refers to the header of a constructor and the class instantiated is assignment compatible with the type of the indirect anaphor, indirect anaphora will be established as described in section 5.3.
  - b) If the anchor refers to the header of a class or interface, the class or interface has a frame that is the conceptual scope of the header. If the frame contains a field whose type is assignment compatible with the result type of the indirect anaphor or if the

---

<sup>14</sup> This is an arbitrary decision that might be waived later, when primitives can be used as anchors.

<sup>15</sup> This is due to the fact that the current implementation inserts a local variable before the statement containing the anchor. This is not possible for explicit constructor invocations that can only be the first statement in a constructor body.

frame contains an accessor method that is compatible with the result type of the indirect anaphor, indirect anaphora will be created as described in section 5.4.

- c) If the anchor refers to the header of a class or interface the frame of which is not suitable for anchoring according to the previous point, indirect anaphora might be inferred as described in section 5.5.

## 5.2 General properties of indirect anaphora in Java

Irregardless of how indirect anaphors are anchored, they share some properties that are outlined in this section.

### 5.2.1 Referential ambiguity

Referential ambiguity means that there is more than one suitable indirect anaphora relation because more than one suitable anchor is available or a single suitable anchor provides access to more than one suitable referent of the indirect anaphor. It is important to keep a compile-time perspective, though: it does not matter in anaphora resolution what the result of a referent represents at runtime. This differentiates referential ambiguity from the alias problem that arises due to the values that variables hold at run-time.

There are cases in which referential ambiguity can be reduced. E.g. when a local variable declaration has an initializing expression and both are suitable anchors for an indirect anaphor, one can safely be discarded.

### 5.2.2 Multiple anaphors per anchor

It is possible that more than one indirect anaphor is anchored in an anchor. In cases when the anchor is a method invocation expression (see below) this introduces a lot of local variables. Future implementations should share these variables.

### 5.2.3 Indirect anaphors as qualifiers

Indirect anaphors can be used to qualify an access to a field or method.

### 5.2.4 Preconditions

For all types of indirect anaphors the following preconditions hold.

1. The anchor is suitable, i.e. it satisfies conditions 1a - 1f, 1h, 1i given on page 30.
2. The indirect anaphor is within the scope of the type used as part of the indirect anaphor (see [GJSB05, 117ff.,132,160ff.,190,263]).

## 5.3 Anchoring based on the headers of invocables

The simplest form of indirect anaphora in Jaaa is that an indirect anaphor relates to a method invocation expression or class instance create expression and denotes the value returned or created by the expression. The pre- and postconditions as well as the algorithm or this anchoring strategy are given below.

### 5.3.1 Preconditions

1. The general preconditions laid out in section 5.2.4 are met.
2. The anchor is a method invocation expression or a class instance creation expression .
3. The anchor refers to the header of an invocable that is accessible<sup>16</sup>.
4. If the anchor is a method invocation expression, the return type of the method invoked is assignment compatible (see [GJSB05, 95]) with the type of the indirect anaphor (i.e. the return type can be assigned to the indirect anaphor).
5. If the anchor is a class instance creation expression, the class of the instance created by the expression must be assignment compatible with the type of the indirect anaphor.
6. The indirect anaphor is not the left-hand side operand of an assignment expression (because the result of the method invocation expression or class instance creation expression denoted by the indirect anaphor is a value).

### 5.3.2 Anchoring algorithm

Indirect anaphora based on invocables could look like in the following snippet, before using indirect anaphora ...

```

52 public static void runMainAndExit(JUnitSystem system,
   String... args) {
53   Result result = new JUnitCore().runMain(system, args);
54   system.exit(result.wasSuccessful() ? 0 : 1);
55 }
```

Listing 5.1: Snippet from org.junit.runner.JUnitCore (JUnit 4.8.2)

... and after ...

```

52 public static void runMainAndExit(JUnitSystem system,
   String... args) {
53   new JUnitCore().runMain(system, args);
```

<sup>16</sup>It is assumed that the method invocation expression or class instance creation expression compiles. For method invocation expressions this means that the method to be invoked is e.g. accessible, applicable and appropriate (see [GJSB05, 442ff.] and [GJSB05, 471ff.]). Such checks will not be reproduced as part of the implementation of this indirect anaphor because they are to be performed as part of the method invocation class instance creation expressions.

## 5 Indirect Anaphora for Java

```
54     system.exit(.Result.wasSuccessful() ? 0 : 1);  
55 }
```

Listing 5.2: The indirect anaphor `.Result` in a modified version of listing 5.1

1. In front of the statement  $S$  containing the expression that acts as the anchor  $A$  of the indirect anaphor  $IA$  insert the declaration of a new local variable  $V$  that has the type used in the  $IA$  and a unique name<sup>17,18</sup>.
2. Replace<sup>19</sup> the method invocation or class instance creation expression<sup>20</sup> that acts as the anchor  $A$  by a parenthetical expression that wraps a cast expression  $C$  that performs a cast to the declared type of the result of  $A$  and contains an assignment expression that assigns the result of  $A$  to the local variable  $V$  introduced in step 1<sup>21</sup>.
3. Rewrite the indirect anaphor into an access to the local variable introduced in step 1.

**Future Work 5.2 (Underspecification of arguments)** *The current implementation of anchoring based on headers of invocables is only able to denote to a return value that has not been stored in a variable. In natural language, however, arbitrary thematic roles of verbs can be omitted – not only results. It would also be desirable to underspecify arguments in method invocations as much as possible, potentially removing all method arguments, or in case of repeated invocations all arguments that remain identical for all invocations. The missing arguments would need to be found from within the block (and potentially its outer blocks, recursively). Most likely the arguments would have to be declared before use - unlike what can be done in natural language. Experiments would need to show how often referential unambiguity can be achieved and how fragile the invocations become to changes of surrounding code in order to evaluate this kind of underspecification.*

---

<sup>17</sup>This step does not yet account for all possible complications two of which have been uncovered already. (1) Within a constructor body, if the implicit or explicit constructor invocation, it must be the first statement [GJSB05, 242f.]. Whether a local variable declaration that is generated by the compiler could precede the constructor invocation is not yet clear to me. At least that might not be the code that bytecode interpreters / JIT compilers expect -> simply disallow use of anaphors in this case, maybe supply an error message. (2) If the statement containing the anchor is the last statement in a block and the indirect anaphor is after the end (outside of) that block, the local variable declaration would need to precede the block, not only the statement containing the anchor.

<sup>18</sup>Note: the `ForInit` and `ForUpdate` parts of a `for` statement might contain an anchor. If that is the case, the entire `for` statement is the one that is rewritten.

<sup>19</sup>The implementation uses `JastAddJ`, a modular compiler that allows a node in the AST to be rewritten into a node of another type or a trees that can contain the original node.

<sup>20</sup>Note that method invocation expressions [GJSB05, 440] as well as class instance creation expressions [GJSB05, 423] have forms that contain a primary expression on the left that can be arbitrarily complex. This modeling is based on the Java language specification, a specific compiler implementation will model the grammar differently to come close to an LL(1) grammar. TODO: Daraus folgt für diesen Schritt?

<sup>21</sup>I did not yet check whether it might be possible that `ClassCastExceptions` are raised at runtime if generics are involved.

## 5.4 Anchoring based on fields and accessors defined by types

The second kind of indirect anaphora in Jaaa implements meronymy-based as well as schema-based anchoring known from sections 2.5.2 and 2.5.3 that allows parts of instances to be accessed.

### 5.4.1 Accessors

Accessors are used to enable other objects access to private fields of an object. There are two kinds of accessors: getters and setters. Their naming is conventionalized (see [GJSB05, 149]).

A *getter* is a method that can simply return the value of a field, or can e.g. initialize the field lazily or convert an internal representation stored in the private field into an external representation provided by the getter. A getter has no parameters and a name that starts with "get" and continues in mixed case, with the first letter after "get" being a capital one. The name is typically derived from the name of the field.

A *setter* is a method that does not return anything (void) and has a single parameter. A setter can simply set the field to the value of the parameter, or e.g. convert an external representation accepted by the setter into an internal one stored in the field. The name of a setter starts with "set" and continues in mixed case, with the first letter after "set" being a capital one. The name is typically derived from the name of the field and the type of the field.

Since the semantics of accessors are only ensured by convention, there are accessors that do not follow the naming conventions. It would be possible to define an annotation that could optionally be added to these unconventional accessors to make them identifiable as accessors.

### 5.4.2 Preconditions

1. The general preconditions laid out in section 5.2.4 are met.
2. The anchor is a local variable declaration, an access to an accessible field, an access to a local variable, parameter, etc.
3. If the indirect anaphor is not the left-hand side operand of an assignment expression and it denotes either a variable or a value and only a single one of the following two cases applies.
  - a) The anchor denotes an expression whose result type declares or inherits exactly one field that is visible, accessible (and static, if the indirect anaphor appears in a static method or initializer) and the type of the field is assignment compatible with the type of the indirect anaphor.
  - b) The anchor denotes an expression whose result type declares or inherits exactly one getter method that is visible, accessible (and static, if the indirect anaphor appears in a static method or initializer) and the type of the indirect anaphor is assignment compatible with the return type of the getter method.

4. If the indirect anaphor is the left-hand side operand of an assignment expression<sup>22</sup> it denotes a variable<sup>23</sup> and only a single one of the following two cases applies.
  - a) The anchor denotes an expression whose result type declares or inherits exactly one field that is visible, accessible, non-final (and static, if the indirect anaphor appears in a static method or initializer) and the type of the indirect anaphor is assignment compatible with the type of the field.
  - b) The anchor denotes an expression whose result type declares or inherits exactly one setter method that is visible, accessible (and static, if the indirect anaphor appears in a static method or initializer) and the type of the indirect anaphor is assignment compatible with the type of the parameter of the setter.

### 5.4.3 Anchoring algorithm

Anchoring based on fields or accessors could look like the following: before ...

```
83     view.getDrawing().fireUndoableEditHappened(edit = new
        CompositeEdit("Punkt verschieben"));
84     Point2D.Double location =
        view.getConstrainer().constrainPoint(
            view.viewToDrawing(getLocation()));
```

Listing 5.3: Snippet from org.jhotdraw.draw.BezierControlPointHandle.java (JHotDraw 7.0.8)

... and after ...

```
83     view.getDrawing().fireUndoableEditHappened(edit = new
        CompositeEdit("Punkt verschieben"));
84     Point2D.Double location = .Constrainer.constrainPoint(
        view.viewToDrawing(getLocation()));
```

Listing 5.4: The indirect anaphor `.Constrainer` in a modified version of listing 5.3

1. If the indirect anaphor denotes a field (see preconditions 3a and 3a) it is replaced by a field access expression that accesses that field on the result of a copy of the expression that serves as the anchor of the indirect anaphor.
2. If the indirect anaphor denotes a getter (see precondition 3b), it is replaced by a method invocation expression invoking that getter on the object that is the result of a copy of the expression serving as the anchor of the indirect anaphor.

<sup>22</sup>Note that an indirect anaphor is not the left-hand side operand of an assignment expression, if it is only a part of the left-hand side operand. Field access expression can take the form `Primary . Identifier` that allows e.g. an indirect anaphor to be used as the contained primary expression. In cases like these, the indirect anaphor is not assigned a value, but the field denoted by the identifier is. In such cases, precondition 3 applies.

<sup>23</sup>This is also required for pre- and postfix increment and decrement operators, but since primitive types are not yet supported (not even indirectly by using unboxing), these operators will be ignored.

3. If the indirect anaphor denotes a setter (S, see precondition 4b), it is the left-hand side operand of an assignment expression (AE) that has a right-hand side operand (RO). The following steps are taken.
  - a) A new private final method (M) is declared that has a unique name and is marked as synthetic. The method M has a single parameter that has the type of the indirect anaphor. The return type of M is the type of the indirect anaphor as well. The body of M contains (1) a method invocation expression that invokes S with the argument of M and (2) thereafter returns the value of M's parameter.
  - b) AE is replaced by a method invocation expression invoking M on the object that is the result of a copy of the expression that serves as the anchor of the indirect anaphor, providing the result of RO as an argument to M.

## 5.5 Inference-based anchoring

Time was up before an implementation of inference-based anchoring could be started. The idea is that for inferences-based indirect anaphors, methods are generated by the compiler that traverse accessors to retrieve the inferred referent. Implementation of thematic progression is assumed to require an implementation of thematic progression, focus and activity.

## 5.6 Test case nomenclature

The implementation was developed test-first with JUnit test cases 70 of which have been developed that are organized using the nomenclature given in table 5.2.

File	Group start and end	Focus of the tests in the group
General	01 - 07	General tests on what expressions can be a suitable anchor
	10 - 19a	Types used as part of indirect anaphors
	20	Referential ambiguity due to multiple occurrences of a potential anchor (see point 1f on page 31 and section 5.2.1)
	30 - 32	The 2-statements limit (see point 1i on page 31)
	40 - 46	Assignment compatibility of the result type of the anchor to the result type of the indirect anaphor (see section 5.1.4)
	50 - 53	Multiple anaphors per anchor (see section 5.2.2)
Type1	01 - 05	Expressions involving method access of various complexity referred to by indirect anaphors of type 1
	20 - 22	Expressions involving class instance creation of various complexity referred to by indirect anaphors of type 1
	30 - 33	Location of indirect anaphor
	40 - 47b	Indirect anaphors used to qualify access to a field or a method
	60 - 61	Exceptions
	80 - 82	Uniqueness of the names that are used for internally generated local variables
	90 - 96	Using indirect anaphors as or as part of the left- or right-hand side operand of assignments or to initialize a local variable
Type2	01 - 02	Retrieving and setting the value of fields
	10 - 11	Referential ambiguity due to concurrent suitability of anchors for type 1

Table 5.2: Grouping of the test cases supplied with the implementation

## 6 Summary and Conclusions

Definitions of reference, proper names and anaphora have been introduced from natural language. More importantly, the perspective of cognitive semantics was adopted that provide models of the structures and processes involved in anaphora resolution. These models were used as a basis the implementation of anaphora resolution in later parts of this work and will continue to guide future implementation work. My study of the literature in cognitive linguistics has been partial only, thought. Besides the fact that I lack a general overview of the field, topics like deixis, direct anaphora, instantiation and specialization of nodes in text-world models as well as the forms of knowledge representation require further reading, as do issues like the depth of conceptual decomposition and thematic progression that are open research questions in linguistics, though.

The relations between natural languages and programming languages were outlined, of which the metaphorical one – even though it is considered hypothetical for the past – is used as the basis of this work. Naturalistic Programming was shown to be along the lines of that metaphorical relation and criticism of usage of the term *natural* was added.

Existing means of reference implemented in Java have been analyzed using terminology from linguistics. One outcome of the analysis was to highlight that names in Java that resemble words from natural language embody a semantic gap caused by the fact that only the programmer has access to the meaning from natural language. The semantic gap can potentially confuse programmers. It was specified that indirect anaphors in Java shall resemble words from natural language only carefully and that anchors shall be freely positionable.

This thesis is obviously not a finished piece of work.

## 6 *Summary and Conclusions*

# Bibliography

- [Bus98] Carsten Busch. *Metaphern in der Informatik*. Deutscher Universitäts-Verlag, 1998.
- [Cho57] Noam Chomsky. *Syntactic Structures*. Mouton & Co., The Hague, Paris, 1957.
- [Cla75] Herbert H. Clark. Bridging. In *Proceedings of the 1975 workshop on Theoretical issues in natural language processing*, TINLAP '75, pages 169–174, Morristown, NJ, USA, 1975. Association for Computational Linguistics.
- [Con04] Manfred Consten. *Anaphorisch oder deiktisch?: Zu einem integrativen Modell domänengebundener Referenz*. Niemeyer, Tübingen, 2004.
- [Coo07] William R. Cook. Applescript. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 1–1–1–21, New York, NY, USA, 2007. ACM.
- [Dic11a] Oxford Dictionaries. <http://oxforddictionaries.com/definition/naturalistic>, last checked 14 April 2011.
- [Dic11b] Oxford Dictionaries. <http://oxforddictionaries.com/definition/natural>, last checked 10 May 2011.
- [Dij78] E.W. Dijkstra. On the foolishness of 'natural language programming'. *Unpublished Report*, 1978.
- [FBP05] James Fan, Ken Barker, and Bruce Porter. Indirect anaphora resolution as semantic path search. In *Proceedings of the 3rd international conference on Knowledge capture*, K-CAP '05, pages 153–160, New York, NY, USA, 2005. ACM.
- [GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison Wesley, 3 edition, 2005.
- [Hen08] Jördis Hensen. *Die Integration von Referenzierungsmechanismen der natürlichen Sprache in Programmiersprachen*. Diploma thesis, Technische Universität Darmstadt, October 2008.
- [HH76] M. A. K. Halliday and Ruaiqiya Hasan. *Cohesion in English*. Longman, Harlow, 1976.
- [Hig67] Bryan Higman. *A comparative study of programming languages*. Macdonald, London, 1967.

## Bibliography

- [Hil65] I. D. Hill. Some remarks on algol 60. *ALGOL Bulletin*, (21):70–74, November 1965.
- [KM06] Roman Knöll and Mira Mezini. Pegasus: first steps toward a naturalistic programming language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 542–559, New York, NY, USA, 2006. ACM.
- [Lan07] Willy van Langendonck. *Theory and Typology of Proper Names*. Mouton de Gruyter, Berlin, 2007.
- [LDLL03] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond aop: toward naturalistic programming. *SIGPLAN Not.*, 38(12):34–43, 2003.
- [LL05] Hugo Liu and Henry Lieberman. Metafor: visualizing stories as code. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 305–307, New York, NY, USA, 2005. ACM.
- [Mit02] Ruslan Mitkov. *Anaphora Resolution*. Longman, London etc., 2002.
- [Nau92] Peter Naur. *Computing: A Human Activity*, chapter Programming Languages, Natural Languages and Mathematics (1975), pages 22–36. ACM Press, New York, 1992.
- [PMMH04] Massimo Poesio, Rahul Mehta, Axel Maroudas, and Janet Hitzeman. Learning to resolve bridging references. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics, ACL '04*, Morristown, NJ, USA, 2004. Association for Computational Linguistics.
- [Ras00] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional, 2000.
- [Sae03] John I. Saeed. *Semantics*. Blackwell, Oxford etc., 2. edition, 2003.
- [Sam66] Jean E. Sammet. The use of english as a programming language. *Commun. ACM*, 9(3):228–230, 1966.
- [Sch00] Monika Schwarz. *Indirekte Anaphern in Texten*. Niemeyer, Tübingen, 2000.
- [Sch08] Monika Schwarz. *Einführung in die Kognitive Linguistik*. A. Francke, Tübingen and Basel, 3rd edition, 2008.
- [SF07] Monika Schwarz-Fiesel. Indirect anaphora in text: A cognitive account. In Monika Schwarz-Fiesel, Manfred Consten, and Mareile Knees, editors, *Anaphors in Text: cognitive, formal and applied approaches to anaphoric reference*, volume 86 of *Studies in Language Companion Series*, pages 3–20. John Benjamins, Amsterdam, 2007.

- [Shn80] Ben Shneiderman. *Software Psychology*. Winthrop, Cambridge, Massachusetts, 1980.
- [Sta09] Daniel Staesche. *Rava – Naturalistic References in Java*. Bachelor’s thesis, Technische Universität Darmstadt, August 2009.
- [SWC<sup>+</sup>95] Neil A. Stillings, Steven E. Weisler, Christopher H. Chase, Mark H. Feinstein, Jay L. Garfield, and Edwina L. Rissland. *Cognitive Science: An Introduction*. MIT Press, 2. edition, 1995.
- [Zem66] H. Zemanek. Semiotics and programming languages. *Commun. ACM*, 9(3):139–143, March 1966.

## *Bibliography*

# Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Berlin, May 11, 2011

Sebastian Lohmeier