Continuing to Shape Statically Resolved Indirect Anaphora for Naturalistic Programming

A transfer from cognitive linguistics to the Java programming language

Sebastian Lohmeier

Contents

List of Future Work

Preface

This is the successor to my submitted but abrupt bachelor thesis. While the implementation is at the same stage as in the thesis, the text was elaborated and corrected for this version and has a number of ideas that had not been mentioned or not been detailed in the thesis. While I hope that the text is intelligible, both the text and the implementation are just a start and will be subject to future modification and extension. However, I wanted to get this version out to be able to get feedback that I did not get while writing my thesis because I am quite reluctant to handing out texts for which even I myself have dozens of points to criticize. If you read this text and have comments about it, please email me. There is also a webpage for this document at http://monochromata.de/shapingIA/ that provides the source code of the implementation described in chapter 7. The page will also inform about new versions of this document when they are available.

My thesis was partly motivated by me reading about Metafor (see [LL05]) a couple of years ago and I meant it to give me an orientation in the field of programming in natural languages or languages that are closer to natural languages. While I had wished for a curriculum that is less like ISDN and more like the Internet instead (in terms of openness, not speed), 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, Daniela Keller, Christian Kolle and chairman Friedrich Steimann who all 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 questions on his work. I thank Susan Segebard who reviewed early parts of my thesis. I am also indepted to my mother who supported me financially at the time when I wrote the thesis and tolerated my limited mood during that time.

Some notes on wording: although it is common in academic texts that a single author refers to himself as *we*, I do not do so. Similarly, I will use the pronoun *one* instead of using *you* as indefinite pronoun.

Sebastian Lohmeier
sl@monochromata.de

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 may do because it is a goal still far from being reached and it is not clear how it could be reached. Others may shiver in anger because they doubt that the goal could be worth striving for. This work 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 aimig 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*, 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[1]. 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, can 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].

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

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[2] 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)* were 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 work 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 work. The work listed in this section I consider closely related. I.e. the works listed here 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[3].

After excluding most related work, two projects remain that have influenced this work, 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 character 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 structual, 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.

[2]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*.

[3]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 have not 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 for 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 models 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 include a detailed discussion of the term *naturalistic programming* which is highly desirable but yet outstanding in the current literature. The project also aims at implementing compile-time resolution of anaphors.

1.2 Aim

Although the idea of programming in natural language or anything closer to it than current programming languages motivated this work, 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 as well as to be able to yield concrete and novel results. The search for literature on linguistics further revealed a model from cognitive linguistics that proved to be central for my work.

In general, I consider this work 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 work, new issues raised are considered part of the outcome of the work. Some of these issues have been highlighted throughout the text in boxes labeled *future work* that are indexed on page vii to make it easier for readers of this work 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 Problem

Now that aims are clear, before discussing the organization of the document, two examples of indirect anaphora taken from later chapters are given to illustrate what problem is to be solved by adding indirect anaphors to Java. The simplest form of indirect anaphora allows the result of a method invocation to be accessed without storing it in a local variable. I.e. instead of storing a result in a local variable as in

```
Result result= new JUnitCore().runMain(system, args);
system.exit(result.wasSuccessful() ? 0 : 1);
```

it can be accessed through the indirect anaphor `.Result`:

```
new JUnitCore().runMain(system, args);
system.exit(.Result.wasSuccessful() ? 0 : 1);
```

A more complex kind of indirect anaphor allows the return value of an accessor to be accessed by providing the type of the return value of the accessor. Instead of invoking the highlighted accessor as in

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

the indirect anaphor `.Constrainer` is used:

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

Both examples will be detailed in chapter 6.

1.4 Organization

This work is interdisciplinary in that it applies cognitive and linguistic concepts to computer science. The chapters reflect this interdisciplinarity through a gradual transition from cognitive 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 are described that are required to understand how humans resolve indirect 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 programing 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.

The fifth chapter integrates the findings from the preceding three chapters into a metaphor of indirect anaphora transferred from natural language to a dialect of Java. The sixth chapter details specific kinds of indirect anaphors and chapter seven describes an implementation of what has been laid out in chapters five and six. The description of the implementation serves as a reference that unlike the other chapters does not close with a summary.

The two final chapters provide an evaluation and an outlook as well as a summary and conclusions.

2 Reference in Natural Languages

Reference is an integral part of natural language[1]. 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 begins 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] [2,3]. For this work it is important to restrict reference to

[1]Although the examples given in this chapter are given in English, the concepts they illustrate can be found in other languages as well. E.g. samples from German were used in [Sch00].

[2]Treatment of reference in linguistics typically ignores explicit references like inter-textual pointers or references to the bibliography of technical texts, its index or cross-references.

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

a relation between linguistic expressions and extra-linguistic entities that is established by the reader [Sch00, 22]. E.g. a name in a sentence can be used to refer to a person. Following Schwarz-Friesel, I will call the the process of establishing reference *referentialization*.

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 functions fulfilled by syntactic entities and that the former refers to sensually perceivable referents and the latter relate within texts (see [Con04, 6] with different terminology). Following this definition, 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) *[Con04] 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). 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 proposed 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 included deixis as well.*

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 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 used 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, the name *C* and the phrase *the Java programming language*[4] are deictic because they refer to the two well-known programming languages that have not 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 can be tightly connected to the referent of that phrase, e.g. in the case of definite descriptions introduced in section 2.3.3. 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 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 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 [Sch00, 59ff.].

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[5].

[4]Use of this phrase cannot be replaced by the proper name *Java* for the latter can be confused with the island named Java, but also with the Java platform that includes the Java Virtual Machine (JVM) for the instruction set of which source code of the Java programming language is typically compiled.

[5]I added subscripted numbers in the example to express that all phrases indexed with the same number share a referent.

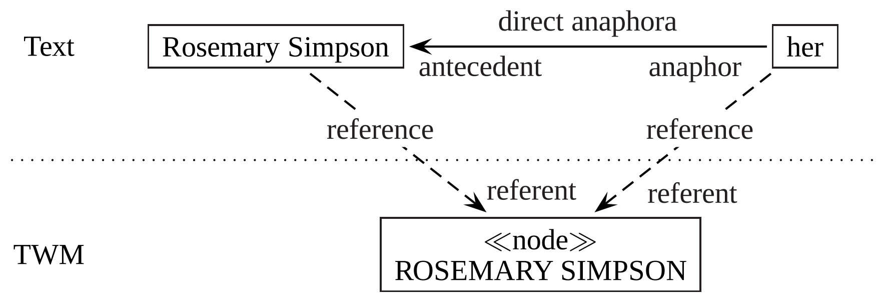

Figure 2.1: Relations in text and text-world model for sample 2.2

Sample 2.2 Rosemary Simpson*1* *worked hard, on* a very tight schedule*2*, *to create* the index*3*. We*4* *got into* the act*5* *at* the last minute*6*, *however; blame* us*4* *and not* her*1* *for* any jokes*7* you*8* *may find hidden* therein*3*. ([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*. The figure 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 an item can be removed from a sentence without hampering understanding of the sentence. The resulting *ellipsis* (depicted as ∅) is 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 (see [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 ∅ remains part of the escaped Unicode stream.* ([GJSB05, 15], ∅ 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 ∅ is treated as a RawInputCharacter and ∅ 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 [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$_1$ *for which it was invoked (that is,* the object$_1$ *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 co-refer to its antecedent *the instance*[6].

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 of Monika Schwarz-Friesel: [Sch00][7]. While [Mit02], [Cla75] and [HH76] were also considered, Schwarz-Friesel's work was more useful due to its cognitive and process-oriented perspective and its complex analysis of indirect anaphora[8]. 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). [Sch00] 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. It did, however, become clear at a later stage of my work that it is necessary to implement direct anaphora along with indirect anaphora.*

[6]The antecedent *the instance* is actually referring as well, as is signalled by the definite article. It may be regarded an indirect anaphor anchored in the indefinite noun phrase *an instance method*.

[7] Some aspects of [Sch00] are summarized in [SF07].

[8]There are also works from computational linguistics that deal with indirect anaphora (see [PMMH04], [FBP05]). These work are, 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 in Java.

2.4.1 Mental representations of knowledge

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*, although 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, the latter of which will be explained later on). The entries of the lexicon describe the core meaning of a word [Sch00, 32]. The lexical meaning of a word is underspecified and independent from context [Sch00, 38], but language-specific [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. Variables can have a default value and 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$^+$95, 159], emphasis in original). A script, Stillings et al. write, "is an elaborate causal chain about a stereotypical event. It can be thought of as a kind of frame where the slots represent ingredient events that are typically in a particular sequence." [SWC$^+$95, 161]. Schwarz-Friesel mentions that scripts are augmented with roles, properties, as well as pre- and post-conditions [Sch00, 35][9].

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 may 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* [Sch00, 38].

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 created by

[9]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 affects the models made up in the field.

the reader while receiving a text. The model 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 model 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 a TWM, three-tier semantics are well suited. "Three-tier semantics distinguish amodal concepts, language-specific lexical meanings and current meanings determined by context" [Sch08, 64, translation mine]. 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, the lexicon entries and conceptual schemata (see [Sch08, 189]). By separating nodes in the TWM from lexical-semantic and conceptual knowledge it becomes possible to model e.g. a text about two different trees without needing to create a sub-concept tree for each tree talked about because each tree has a node in the TWM and these nodes are derived from the concept of a tree. Through 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. Referentialization is part of the elaboration of the propositions that are the semantic content of a text. Elaboration integrates conceptual knowledge from the reader's memory into the TWM by means of cognitive strategies. Note that the initial TWM is described as being made up of propositions contained in the text and then elaborated instead of strictly distinguishing propositions and TWM as is done in other theories (see [Sch08, 197]). 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 nodes 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 from the text – is described in the literature and will become more relevant to more complex uses of anaphora. [Sch08, 64f.,189f.] may be good starting points; it may actually be a good idea to read the entire book (for me too, I found it quite late during my 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. This 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 referring to the node had been read. The node can 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.

Anchor instead of antecedent "The previous text does not contain an explicit *antecedent* but rather an [...] *anchor*, that is essential for the interpretation of the indirect anaphor." [Sch00, 50, translation mine]

Conceptual relation "The referents of anchor and indirect anaphor do not stand in a co-reference relation but in another close, conceptual relation." [Sch00, 50, translation mine]

Constructive interpretation "Interpretation of indirect anaphors includes constructive activation of knowledge on the part of the recipient instead of a mere process of searching and matching." [Sch00, 50, translation mine]

No demonstratives or pronouns Demonstratives and pronouns can only in rare cases be used as indirect anaphors. [Sch00, 50]

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 (see [Sch00, 82]). Underspecification may hinder referentialization, if the reader does not possess the knowledge anticipated by the writer. It is, however, frequently[10] used because language is used economically [Sch00, 83].

Future Work 2.5 (Research on indirect anaphors in technical texts) *It would be interesting to know how frequent indirect anaphors are in 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. While this evidence is anecdotal, it would be worth a detailed examination: Schwarz-Friesel reports on an experiment of her in which 40 % of the subjects found full specifications superfluous while the corresponding underspecifications were not deemed inappropriate [Sch00, 79f.]. The experiment seems to be based on non-technical texts, though. It would hence be interesting to find existing research on this matter or (1) analyze a corpus of technical texts for occurrences of indirect anaphora and (2) conduct an experiment to show under what circumstances a subject deems full specification superfluous respectively deems underspecification inappropriate.*

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 exemplify 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 (verb arguments 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 (⚓), as can be seen in the example below.

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

Figure 2.2 illustrates[11] 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*. All other lexicon entries are omitted. 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 although it

[10]Schwarz-Friesel actually quotes studies based on corpora of Swedish and English texts that revealed that about 60 % of definite noun phrases have no explicit antecedent (see [Sch00, 79]).

[11] The figure is an illustration in the sense that depicts entities of a theory, and that I do not yet know how text-world models, lexicon entries (and in later illustrations) conceptual schemata are *commonly* depicted in cognitive linguistics (it may be that there is no consent on their depiction). The illustration is based on the graphical elements of the Unified Modeling Language, but does not adhere to the syntax and semantics of UML.

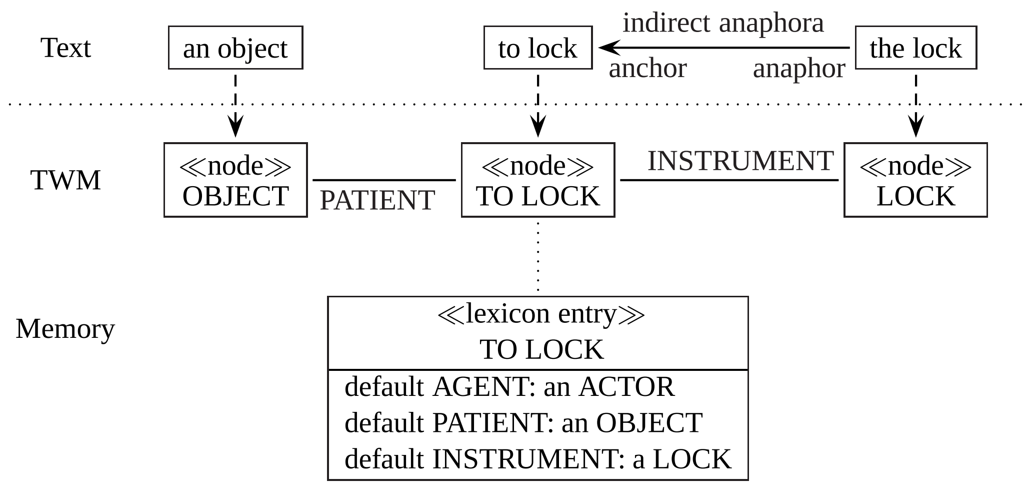

Figure 2.2: Relations in text and text-world model for sample 2.6

has not been mentioned before[12]. The other definite noun phrases of the sample are no examples of indirect anaphors based on verb semantics only[13]. 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 related to the node of *to lock* in the TWM. 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 [14,15].

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 1*$_{PATIENT(1)}$ *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)

[12]Had it been introduced via the indefinite noun phrase *a lock* before, *the lock* would be a direct anaphor because it would co-refer to the antecedent *a lock*.

[13](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.

[14]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.

[15]This example also shows the cohesive force of indirect anaphora: the verb-semantical relationship between *to lock* and *the lock* spans the two sentences, turning them into a coherent chunk of text.

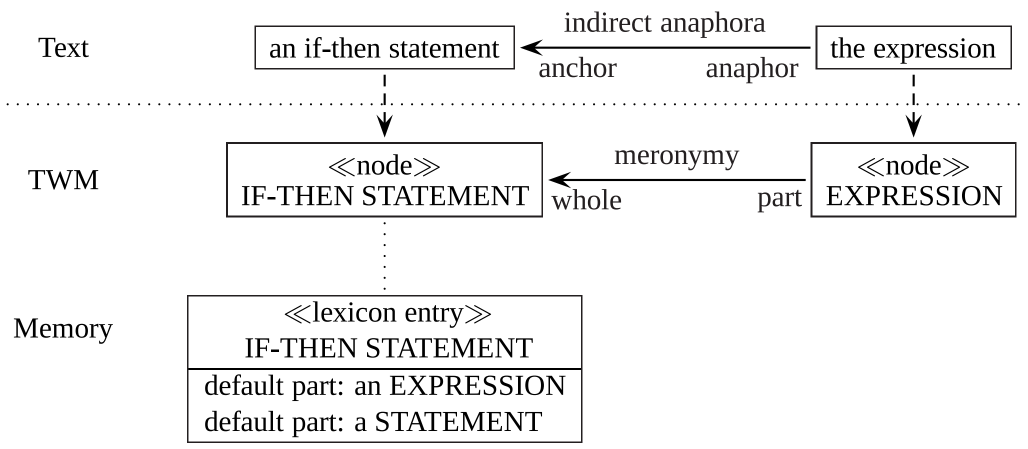

Figure 2.3: Relations in text and text-world model for sample 2.8

The indirect anaphor *the sum* takes the 3rd PATIENT[16] role of the verb *add* that appears in the verb phrase *is added* in the first sentence. This case is different from the first one, in that not only knowledge from the verb's semantic entry in the mental lexicon and the lexicon entry of the indirect anaphor are involved in the resolution of the indirect anaphora. The verb *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 firstly to make the sentence sound[17]. 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 node in the TWM. I.e. indirect anaphora based on thematic roles may not only involve semantic knowledge but also conceptual knowledge.

2.5.2 Meronymy-based anchoring

Not only thematic roles of verbs are modeled as part of the mental lexicon, the lexicon 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 11. 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 (see [Sch00, 104ff.]). Meronymy is the name used for part-whole- and similar relations, as in

[16]It becomes obvious here that generic models of thematic roles have their limitations: it is hard to classify roles involved in abstract processes. From a programming perspective it is also relevant that at least Saeed does not list a role for the outcome of an action that could be used for creative processes or calculations.

[17]**Otherwise, the pine tree is added to the garden and the sum ...* would have been invalid because sums have nothing to do with gardening.

the following example.

Sample 2.8 *An if-then statement*⚓$_{(1)}$ *is executed by first evaluating the Expression*$_{IA(1),}$⚓$_{(2)}$*. 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[18]. It is also expected that the reader knows that an expression is evaluated, yielding a value that is the result of the evaluation and is, like all other values in Java, typed. Theoretical entities involved when a reader reads the sample are illustrated in figure 2.8. A reader who encounters the given sample can see from its indefinite article, that the noun phrase *an if-then statement* is a new entity referred to by the text and a new node needs to be constructed in the TWM, which acts as the referent of the phrase. The lexicon entry associated to the newly created node contains defaults for the parts of an 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 even though no corresponding node exists in the TWM yet. Since no expression has been introduced before, a new node EXPRESSION is created that is the referent of *the Expression*. The node IF-THEN STATEMENT is active in the TWM and its associated lexicon entry has a default for an EXPRESSION which is now replaced by the newly created EXPRESSION node taking a meronymic relation to the IF-THEN STATEMENT. This relation in the TWM reflects the indirect anaphora relation that is expressed in the text. Similar to the anchoring just described, *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 kind of anaphora, e.g. in the case of the initial sentence of the sample by rewriting it to "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[19].

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 phrase *by first evaluating 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.

2.5.3 Schema-based anchoring

There can be cases in which the entry in the mental lexicon that a node of an anchor is based on is not sufficient to establish a 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,

[18]In the original text, *expression* appeared capitalized and italicized to highlight that the word refers to the preceding grammar definition.

[19]This form is actually used frequently in the Java language specification.

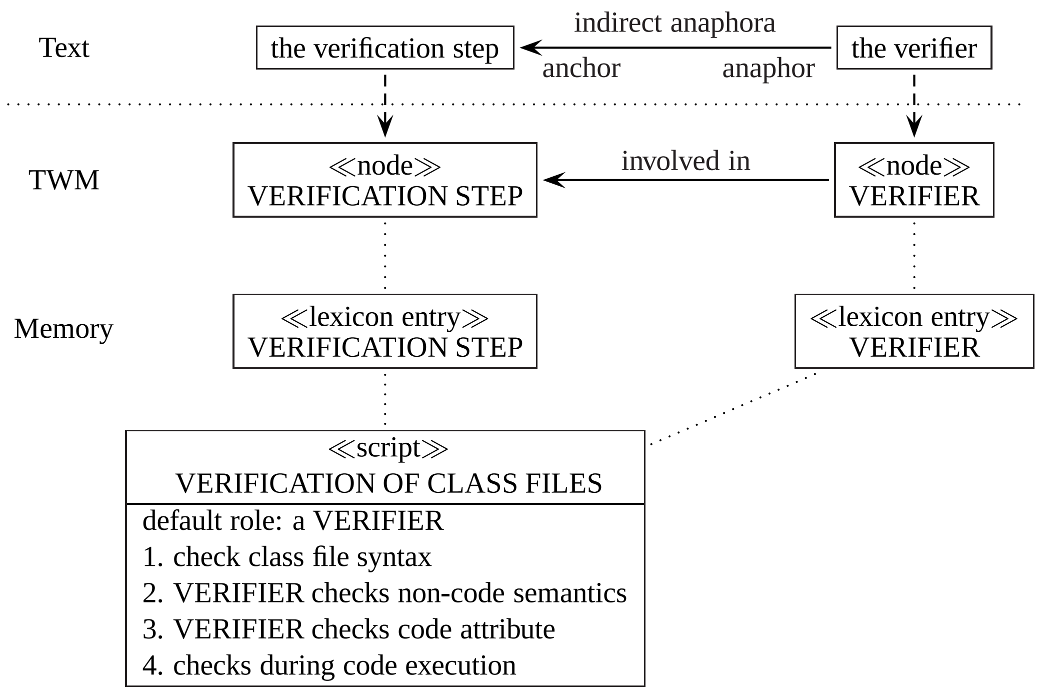
Text
the verification step
indirect anaphora
anchor
anaphor
the verifier
TWM
≪node≫ VERIFICATION STEP
involved in
≪node≫ VERIFIER
Memory
≪lexicon entry≫ VERIFICATION STEP
≪lexicon entry≫ VERIFIER
≪script≫ VERIFICATION OF CLASS FILES
default role: a VERIFIER
1. check class file syntax
2. VERIFIER checks non-code semantics
3. VERIFIER checks code attribute
4. checks during code execution

Figure 2.4: Relations in text and text-world model for sample 2.9

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 ⚓ : the program might run and print:*
s
This demonstrates that without the verifier$_{IA}$ the type system could be defeated by linking inconsistent binary files ([GJSB05, 342], markup mine)

Unlike previous samples, the discussion of this sample will initially ignore the text surrounding the sample in the Java language specification. The theoretical entities involved when a reader is reading the sample are illustrated in figure 2.4. The entries of the mental lexicon and the contents of the script shown in the figure are at this point considered given.

Based on above simplifying assumptions, the resolution of the indirect anaphor *the verifier* can be explained as follows. Reading the definite noun phrase *the verification step* creates a new node and semi-activates the script VERIFICATION OF CLASS FILES[20]. When the phrase *the verifier* is read, a new node VERIFIER is created. Because the script has a default role for a VERIFIER, an *involved in* relation is created between the VERIFICATION STEP and the VERIFIER node in the TWM parallelling the indirect anaphora relation. Note that *the verifier*

[20]The fact that determiner *the* signals that a node already exists is treated later.

is not a part of *the verification step* and is therefore not part of the latter's semantics that may be stored in the mental lexicon.

The contents of the script VERIFICATION OF CLASS FILES has been considered given but will now be discussed. Each script must have been created somehow. While I did not read the literature to find out about models of how concepts and conceptual schemata are created, I will for now assume that scripts are created by reading texts describing processes. The Java language specification itself does not describe the process of class file verification. The script given is a superficial outline of the relevant section of the Java Virtual Machine (JVM) specification (see [LY99, 141ff.]). I.e. the assumption underlying the given script is that the reader modeled in figure 2.4 read the JVM specification before reading the Java language specification and recalls those details from the JVM specification that are given in the script.

While it may also be possible that a reader has a more elaborate script on the VERIFICATION OF CLASS FILES, there may also be readers who do not know such a script at all. Can they be able to anchor the indirect anaphor? Even a reader who does not know what *verification* means can create a minimal script by applying her (implicit) knowledge of morphology to the words *verification* and *verifier*. The reader may have a more vague idea than put forward, but he is likely able to identify that *verification* and *verifier* are related to the verb to *verify* and that the suffix *-ation* in *verification* expresses a process and the suffix *-er* in *verifier* expresses a human or abstract actor performing a verification. Based on such potentially unconscious analysis the reader would be able to create an ad-hoc script that may be named VERIFICATION and contain nothing but a VERIFIER in a role default and is used to anchor the anaphor.

Future Work 2.6 (Creation of concepts and conceptual schemata) *The literature shall be searched for models describing the creation of concepts and conceptual schemata during the reading of a text. For the idea of three-tier semantics it is especially important how nodes in the TWM are turned into concepts and conceptual schemata - i.e. become part of the reader's knowledge, so they can be incorporated in the understanding of other texts.*

Not only the contents of the script can be objected, the form of the entries in the mental lexicon as shown in figure 2.4 can be as well. That the VERIFICATION STEP node is created from a single lexicon entry for VERIFICATION STEP is plausible if that is e.g. a technical term repeatedly used in the literature. That is, however, not the case in this instance. Referring to a verification *step* does instead suggest that verification is part of a process. The anaphoric use of *the verification step* does thus lead to the question how nodes in the TWM can be constructed based on more than one lexicon entry. Answering this question requires further reading.

Future Work 2.7 (Involvement of entries from the mental lexicon) *My reading was yet insufficient for detailing the grain of lexicon entries as well as whether and if so, how, multiple lexicon entries can be involved in the creation or elaboration of a node in the TWM.*

The phrase *the verification step* is relevant for another aspect as well: the sample was quoted out of context and I did so far ignore the fact that the determiner *the* in *the verification step* signals that a node suitable for this phrase is already present in the TWM. Hence, that node must have been introduced in the pre-text or an anchor must have been available in the pre-text the beginning of which is shown in the following sample.

Sample 2.10 *This version of class Super is not a subclass of Hyper. If we then run the existing binaries of Hyper and Test with the new version of Super, then a VerifyError is thrown at link time. The verifier objects [...].* ([GJSB05, 342], markup mine)

It becomes clear that *the verification step* itself is a schema-based indirect anaphor with *link time* being the corresponding anchor - verification is one step in linking (see [GJSB05, 310]). The pre-text also contains the initial mention of *the verifier* in the entire Java language specification, hence that one is the actual schema-based indirect anaphor if the entire text is read sequentially. The subsequent mention of *the verifier* in sample 2.9 is, when the pre-text is read as well, only a direct anaphor referring again to an already existing and active node in the TWM. Which entities are involved in the creation of the node for *the verifier* in the pre-text is less clear than in sample 2.9 though, because the pre-text also mentions *a VerifyError* besides *link time*, giving two potential anchors that trigger the semi-activation of the script containing a default VERIFIER that is replaced by the referent of the indirect anaphor *the verifier*. This exemplifies how hard it was to find clear examples for schema-based anchoring for most potential samples involved meronymy, verb-semantical roles or inference.

While the discussion of samples 2.9 and 2.10 exemplified that an indirect anaphor is a context-sensitive and reader-related function that a phrase can have, the discussion revealed that multiple phrases can be suitable anchors for an indirect anaphor and that further reading is required.

In sample 2.9, the definite noun phrase *the type system* has not yet been discussed. It appears to be less connected to CLASS VERIFICATION and seems to be an example of *complex anaphora* that will not yet be treated here.

Future Work 2.8 (Cover complex anaphors) *Schwarz-Friesel worked on complex anaphors that have been mentioned in [Sch00, 129ff.] and [Sch08, 199ff.] and have been the subject of a research project whose artifacts are listed at http://www.coling-uni-jena.de/ig-wiki/index.php/Prof._Dr._Monika_Schwarz-Friesel/Komplextex . Complex anaphora is a form of anaphora that allows a single anaphor refer to "a complex linguistic entity, which means that it consists of (at least) a clause" [CKSF07, 83] and is subject to further constraints. The ability to refer to complex referents makes this kind of anaphora an interesting candidate for another transfer to programming languages.*

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 and thereby exceeds what can be done solely based on semantic knowledge (see [Sch00, 89]). Inference of indirect anaphora is limited by three factors: "the semantic representation of the definite phrase being used as indirect anaphor, the anchor and the existing TWM" [Sch00, 90, translation mine].

Hence, inference-based indirect anaphora (see [Sch00, 114ff.]) are those kinds of anaphora for which not only a default from a conceptual schema needs to be replaced in order to establish them. Schwarz-Friesel distinguishes two kinds of inference-based anchoring[21] (see [Sch00, 89]):

[21]This is not meant to say that there are no boundary cases.

1. Activative inference: Conceptual schemata are activated in order to disambiguate, specify or elaborate the semantics of a text.

2. Constructive inference: Conceptual schemata are activated in order to create new mental representations (e.g. concepts, schemata, nodes in the TWM) that are used to construct the TWM.

Since inference-based anaphors depend upon the reader's knowledge to an even greater extent than schema-based anaphors do, the discussion of an example is even more hypothetical (and superficial). Since the transfer of this form of indirect anaphora has not yet been started, this section serves as a placeholder indicating that there is yet another form of indirect anaphora to be considered. A detailed description will be worked out when an implementation is attempted. Having said this, the following sample contains an inference-based indirect anaphor.

Sample 2.11 *A program terminates all its activity and exits when one of two things happens:*

- *All the threads that are not daemon threads terminate.*
- *Some thread*$_{ACTOR}$ *invokes*$_{⚓}$ *the exit method of class Runtime or class System*$_{THEME}$ *and the exit operation*$_{CA}$ *is not forbidden by the security manager*$_{IA}$.

([GJSB05, 331], markup mine)

The last sentence is of interest here. It links *the exit method* to the classes that provide it. In contrast, *the security manager* has a definite article as well, but has not been introduced before. It is anchored in the METHOD INVOCATION script which is the conceptual scope of *invoke* and its ACTOR and THEME roles given in the text. Whether activative or constructive inference-based anchoring happens depends on the knowledge of the reader. If the reader knows that a method may upon invocation query the security manager to ensure that the invocation is permitted, activative inference is possible. The verb phrase *is not forbidden* will then hint the reader at the fact that METHODS CAN QUERY THE SECURITY MANAGER UPON INVOCATION TO MAKE SURE THE INVOCATION IS VALID. Since the fact is known to the reader, it can be integrated into the TWM to establish indirect anaphora between the anchor METHOD INVOCATION and the anaphor *the security manager*. It would be worth considering constructive inference that is necessary when the reader does not possess the required knowledge. I know too little about inference, however, to be able to provide a good description for this sample yet. Besides that, there are a number of properties of the sample that complicate a description of the inference: (1) the sample states a condition, (2) it includes a negation as well as (3) the complex anaphor *the exit operation* that refers to *a program terminates all its activity and exits* and (4) the propositional structure made up from *the exit operation is not forbidden by the security manager* does already allow a TWM to be created that involves *the security manager* even though a conceptual relation is still missing.

Future Work 2.9 (Further understanding of inference) *Schwarz-Friesel's treatment of inference does not yet allow for a straight implementation of inference-based anchoring. A number of points need clarification. (1) The samples in [Sch00, 114ff.] are not discussed in such a detail* *that would make clear all steps and conditionals of the inference process. [Sch00, 88ff.] provides references to further literature, though. Besides that, most of the samples seem to be cases of constructive inference. (2) The relation of the concept of inference that Schwarz-Friesel uses to the ones used in logic (i.e. deduction, induction and abduction) are unclear to me. They seem to be relevant, though, because to interpret Schwarz-Friesel's samples, information needs to be* assumed *as in the case of induction or abduction. In this respect it would also be interesting to find which forms of (logical) inference apply, since it seems less desirable to transfer means of reference to programming languages that are uncertain (like induction and abduction can be).*

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 that is reflected by a relation 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 by Schwarz-Friesel to explain the selection of a suitable anchor from a set of potential anchors. Schwarz-Friesel calls these steps *cognitive strategies* for they happen automatically and unconsciously [Sch00, 135].

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 definite noun phrase which is to function as an indirect anaphor. [Sch00, 135] I integrate into this step textual factors that Schwarz-Friesel details later, when she states that the search for a suitable anchor requires that anchor and indirect anaphor are reasonably close to each other within the text and that such a suitable anchor is typically[22] determined by the following conditions (see [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 anchor's cognitive domain can be set by a referent of 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) Involvement of focused theme: When the indirect anaphor is processed on-line, the cognitive domain of the potential anchor must be part of the currently focused theme in order to allow for a relation between anchor and indirect anaphor to be established easily (it is not yet clear what a theme is, though, see future work item 2.10 below). 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 2.10 below.

[22]i.e. the conditions may be incomplete or overly restrictive in exceptional cases

2. "Establish the relation between indirect anaphor and the suitable anchor by either

    a) searching the cognitive domain of the anchor for a semantic role that the indirect anaphor can take[23], or

    b) searching the cognitive domain of the anchor for a conceptual role that the indirect anaphor can take, or

    c) if there is no role explicitly stored in the mental lexicon or concept memory, using inference to create a role in the TWM that is taken by the indirect anaphor." [Sch00, 135, translation and footnote mine]

   Schwarz-Friesel expresses that she takes a "(moderate) minimalist" position concerning when to use text-semantic knowledge 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 2.10 (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 and 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 also asserts that there may be multiple themes that are gradually activated. While she points out that theme and rheme are distinguished by how easily they are reachable in memory [Sch00, 92], and that a theme is focused [Sch00, 142] i.e. a rheme can only be activated, she does not detail how a theme or rheme is represented. Instead, she describes them as "mental values of information that are assigned to linguistic representations of varied complexity" [Sch00, 92, translation mine]. I.e. it remains unclear what values of information and linguistic representations of varied complexity are and how both are mapped onto each other. It seems likely that the granularity of themes determines when a switch of theme occurs. If one assumes that conceptual schemata function as themes or are at least related to themes, their granularity becomes relevant for thematic progression. Unfortunately, Schwarz-Friesel points out that it is unclear how fine-grained conceptual schemata are (see [Sch00, 35]). I.e. 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. This shows that the idea of a* theme *needs further clarification to be found in the literature in order to effectively determine the scope searched for potential anchors of an indirect anaphor.*

[23]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

2.6 Summary

In this chapter, reference, anaphora, proper and common names have been defined and the need to deal with deixis has been pointed out. A modular theory of mental representations of semantics was introduced that distinguishes a mental lexicon and conceptual schemata and explains their interconnections, even though the granularity of concepts is yet unclear. Three-tier semantics were outlined that involve conceptual and semantic knowledge as well as a text-world model and how readers elaborate this model. However, instantiation and specification of nodes of the TWM is not yet fully covered. Additionally, focus and activity were described, that serve to explain the dynamic limitation of searches for antecedents. Anaphora was subclassified into direct and indirect anaphora. The latter has been exemplified, but the part of Schwarz-Friesel's model treating direct anaphora has not yet been integrated here. Even though data on indirect anaphors in technical texts is outstanding, four kinds of indirect anaphora were distinguished, based on the knowledge involved in their resolution. The first kind is based on thematic roles, but may involve additional conceptual knowledge. The second kind is based on (intransitive) meronymy relations. In the discussion of the third, schema-based kind, the extent to which understanding depends on the individual reader became an issue besides the need to cover the (ad-hoc) creation of schemata and the involvement of more than one lexicon entry. Finally, indirect anaphors based on activative and constructive inference have been distinguished, even though the latter one could not be exemplified, since a detailed description of the inference process and its relation to logical inference it still to be found in the literature. Another form of anaphors (complex anaphors) is yet to be treated. In the last section, an algorithm for selecting a single suitable anchor from a set of potential anchors was reproduced that involves the economical selection of the kind of anchoring to be applied. The algorithm includes thematic progression which is based on focus and activity but requires further reading to clarify the representation and granularity of theme and rheme.

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 may 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 neccessarily 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 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 *intuitive* and *natural*, when used to describe user interfaces, mean that something is known or easy to learn [Ras00, 150-1]. It may 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 occurence in natural language. This is necessary because not all implementations match the feature of natural language they seek to implement, which is a 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 may also be a lot of evidence for the idea that natural langugage 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 in programming languages, (2) both are sub-concepts of the same superordinate concept of language and (3) they may be metaphorically connected. The metaphorical relation I consider hypothetical for the past but I will pursue it in my own transfer. The transfer will be oriented towards naturalistic programming, even though that term should be subject to criticism.

4 Reference in Java

After analyzing forms of reference in natural language and the relationship between natural language and programming languages, 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, static initializers and instance 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, occurences 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[1]). 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.

[1]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.]).

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, 147]
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 may 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 may be a good application of Consten's gradual scale between deixis and anaphors (see page 8). 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 of natural language (see section 2.3.1)[2]. 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 directly refer to an object that does not appear in the source code, namely `this` and `super`. These forms do neither funcation as direct anaphors, nor do they function as indirect anaphors. The forms do not function as direct anaphors for Java has no notion of gender and number in which the presumed anaphor and its antecedent could agree and there is no antecedent in the text to which the presumed anaphor relates and could be close to (cf. Schwarz-Friesel's canonical conditions mentioned in section 2.3). The forms do not function as 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 (cf. section 2.5).

Java's keyword `this` can be used "in the body of an instance method, instance initializer or constructor, or in the initializer of an instance variable of a class. [...] When used as a

[2]e.g. `this` in Java, `self` in Smalltalk, `me` in VisualBasic

primary expression, the keyword `this` denotes a value that is a reference to the object for which the instance method was invoked (§15.12), or to the object being constructed." [GJSB05, 421, markup in original] It was stated above that this syntactical variant of `this` reflects the deictic use of the English demonstrative pronoun *this*[3]. There is another syntactical variant involving `this` in Java: "Any lexically enclosing instance can be referred to by explicitly qualifying the keyword `this`. [...] The value of an expression of the form *ClassName*`.this` is the *n*th lexically enclosing instance of `this` (§8.1.3)." [GJSB05, 422, markup in original] This variant is similar to the use of *this* as a determiner in an NP[4]. While qualified `this` does in Java appear after the class name used to qualify it, it is the other way around in English: *this* is used as a determiner that precedes a noun in a noun phrase.

It can be concluded that the lexical resemblance between `this` and *this* is met by only partially overlapping semantics. A flexible aspect of *this* that `this` does not cover is the choice of potential referents: `this` is limited to referring to enclosing instances or the currently executing object. Since the limited overlap does not make `this` natural in the sense laid out in section 3.2, new forms of reference should be sought that are modeled after anaphors, i.e. that are related to an antecedent or anchor in the text that can be freely positioned by the user of the programming language as is the case with local variable declarations except for the fact that the antecedent or anchor shall not be explicitly named[5].

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()`. This will lead the compiler to search for a matching method declaration in an enclosing type declaration that is both visible and accessible [GJSB05, 442]. 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

[3]The deictic use of *this* applies e.g. when at a market, asking "What do you think about this?" while pointing at the rhubarb offered. The anaphoric use, that was said to be impossible in Java, allows English to be used as in the following example: "So, for example, when we list the ways in which an object can be created, we generally do not include the ways in which the reflective API can accomplish this." [GJSB05, 6]. Such anaphors have also been called *discourse deixis*.

[4]Such NPs can in English be both deictic and anaphoric. Deictic, e.g. when asking "What do you think about this stalk?" when pointing at a stalk of rhubarb. Anaphoric as in the following example: "Over the past few years, the Java^{TM} programming language has enjoyed unprecedented success. This success has brought a challenge [...]" [GJSB05, xxvii, superscript in original]

[5] 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.

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 identified that clearly results from indirect anaphora. What then could make an indirect anaphor in Java? Schwarz-Friesel identified features of indirect anaphora (see [Sch00, 118]), a number of theses 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 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 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 Constructing a Metaphor

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, instance initializers and static initializers. Based on the analyses from the previous chapters, this metaphor will be developed in an abstract fashion in this chapter. The subsequent chapter will detail specific forms of the metaphor. An implementation of what has been laid out here and in chapter 6 will be described in chapter 7.

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 call *Jaaa* to have a name to refer to it[1]. 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 can 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[2]. 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[3]. Each of the following sections includes a table summarizing the elements of the two contexts compared in the section[4].

[1]Only minimal effort was made to find this name.

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

[3]Upon more detailed inspection than performed here, cases may be identified that let 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.

[4]This version of the document includes references to the test cases that will look like this footnote. Test cases that check a valid result are prefixed by a "+", test cases that check error handling are prefixed by a "-". Each reference to a test case includes the fully qualified name of the class containing the test as well as the method of that class that implements the test case.

5.1 Pragmatics

Element	Natural Language	Jaaa
Participants	Humans, rarely computers	Humans as writers, computers and humans as readers
Sovereign over definition	Technically: none	Compiler
Reading order	Potentially non-linear	Typically non-linear
Text (broad)	Textual material consumed	Source code consumed
Text (narrow)	Coherent sentences	Anaphoric scope: a method declaration, a constructor declaration, an instance initializer or a static initializer

Table 5.1: Elements of pragmatics compared

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[5], but read by humans and compilers. Further, in the latter case compilers have sovereignty of definition over what a valid program is and what semantics programs have.

A compiler is typically instructed to read and translate all supplied source code during a compilation run. In the context of natural language, e.g. novels are texts that are commonly read in their entirety as well. How much of a text is read, is, however, the individual choice of the reader, i.e. the text consumed is different in extent from the text written. This is true for compilers just as much as it is for readers of novels because a compiler may be presented with a large base of source code but may be asked to compile only a fraction of it making the compiler read only a necessary subset of the sources presented. Novels and source code do not only have things in common, but do of course differ as well, not only in the language used. Novels are normally organized for linear reading from beginning to end. Object-oriented source code is, however, due to its aim at re-use, inherently non-linear. While texts typically occur in books, not all books are written for linear reading: dictionaries, encyclopedias as well as cookbooks are collections of small texts that cross-reference each other either explicitly or implicitly to allow for non-linear reading with considerably more entry points for coherent reading than in a novel or a collection of articles.

A broad definition of text may be used to describe the entire textual material consumed - the read parts of a book in the case of natural language or the parsed source code in the case of Java. This broad definition of text will be used to transfer the concept of TWM to Java later on.

A narrow definition of text fits the single articles or recipes in that it is assumed that a text is defined by the coherence established by anaphora and other means, but not by cross-references (see footnote 2 on page 7). This narrow definition of text will serve to limit the scope of indirect

[5]Exceptions to this statement are subsumed under the term *generative programming* that describes the use of computer programs that generate source code.

anaphora whose transfer to Java is the purpose of this chapter. As already stated in the introductory paragraph of this chapter, I will apply indirect anaphora within the bodies of methods (see [GJSB05, 209]), bodies of constructors (see [GJSB05, 240]), bodies of instance initializers (see [GJSB05, 238]) and bodies of static initializers (see [GJSB05, 239]) only[6]. Each declaration of a method, constructor or initializer I will regard as a separate text according to the narrow definition that is unrelated to all other such declarations in a file of source code. I will refer to this narrowly defined text as *anaphoric scope*. Note that while the mentioned declarations include a body which is the only place in which indirect anaphors can occur, the text is constituted by the entire declaration because some potential anchors may lay outside the declaration's body. E.g. methods and constructors declare parameters that an indirect anaphor may be anchored in even though parameter declarations are not part of the body of methods and constructors.

What is the reason for this limitation to the narrow definition of text? The potential that a programmer is aware of an anchor suitable for an indirect anaphor. To be able to recognize a suitable anchor for an indirect anaphor, its node in the TWM must be active, its theme focused (see sections 2.4.3 and 2.5.5). An anchor's node will be active, if the programmer did just read it and its theme will be focused if no other theme became focused meanwhile. I assume that this is likely within bodies of methods, constructors or initializers. It is, however, unlikely outside of them for the following reasons. 1. Linearity applies within method bodies only, there is no reading order defined outside of method bodies – i.e. it is unclear which member declarations of a class declaration are read and in what order. 2. Similarly, inherited members cannot be assumed to be active and focused because it is unlikely that the programmer read the super-type definitions recently. 3. This does apply to accessible members of other used types as well.

5.2 Syntax

Element	Natural Language	Jaaa
Top-level structure	Sentence	Statement
Phrases	NP; VP	Expression; method invocation expression
Pronouns	*this* and many others	`this`, `super`
Ellipsis	Supported	Supported
Indirect anaphor (here)	the <noun>	`IA ::= . Name |` `IA(Arguments`$_{opt}$`)`

Table 5.2: Elements of syntax compared

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

[6]+ tests.Kind1.test30IAInConstructor()
+ tests.Kind1.test31IAInStaticInitializer()
+ tests.Kind1.test32IAInInstanceInitializer()
+ tests.Kind1.test33IAInConstructorNoInterferenceWithInstanceInitializer()

natural language sentences with Java statements. This is plausible because sentences are the highest level syntactic elements in natural languages which is true for statements in bodies of anaphoric scopes as well. Additionally, Java statements can have nested sub-statements just like sentences can have nested sub-sentences that are called clauses.

In natural language, the next smaller syntactic element after clauses are phrases. 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 NPs are typically used to refer while expression are at runtime evaluated to a value which they may be said to refer to. 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 partly covered by the naming conventions of the JLS (see section 4.1). This comparison is inconsistent from the perspective of natural language, though: 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[7].

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 refernt is known to the reader and <*noun*> is a noun hinting at the referent. Analogous to that I will only handle indirect anaphors that take the form `.Name` or `IndirectAnaphor ( Arguments`$_{opt}$ `)` in Jaaa[8]. This can be expressed by the syntax definition in listing 5.1 (The syntax of the definition follows the convention used in the Java language specification (see [GJSB05, 10]).)

```
IndirectAnaphor :
    . Name
    IndirectAnaphor ( Arguments_opt )
```

Listing 5.1: Syntax definition for indirect anaphors in Jaaa

The indirect anaphor is a primary expression (see [GJSB05, 420]) in Jaaa. The initial dot (`.`) of an indirect anaphor acts as definite determiner known from natural language that signals that a referent is available. 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. Using `the` instead of a dot would make the lack of an implementation of direct anaphors even more obvious and thus irritate further. The choice of the dot does have a positive aspect as well: it is used to connect the parts of a chain of field accesses and method invocations and qualified names in Java. Now that it is used at the beginning of indirect anaphors, one could assume that there is something missing in front of the dot, which is true: the information missing in front of

[7] Method invocation expressions for methods that return `void` are an exception to this.

[8] + tests.General.test60AutonomousIA()
+ tests.General.test61IAInMethodAccess()
+ tests.General.test62IAInChainedMethodAccess()
- tests.General.test63TwoIAsChainedInChainedMethodAccess()
+ tests.General.test64IAAndFieldAccessInChainedMethodAccess()

the dot will be derived from the anchor that will be identified and an error will be raised if no suitable anchor is available. Note that indirect anaphors can be used to qualify access to a field or method[9].

5.3 Cognitive Foundations

Element	Natural Language	Jaaa
Resolution happens	While reading	Statically (i.e. at compile-time)
Construct of comprehension	Text-world model	Abstract syntax tree
Knowledge representation	Textual, memorized	Textual = memorized
Kind of model	Schwarz-Friesel: modular	here: modular
Lexicon entry	Entries in mental lexicon	Headers of invocables: parameter and return types; Headers of classes and interfaces: direct and indirect supertypes
Conceptual schema	Frame; Script	Frame of a type: fields, accessors, direct and indirect supertypes; Script (potentially): body of an invocable
Model	TWM	AST=TWM, extra corpus nodes=memory
Text understanding	construction: TWM from text semantics, elaboration: TWM + memory (includes anchoring)	parsing: AST from source, transformation: AST + text (broad + narrow) (includes anchoring)
Elaboration	Typically on-line	Typically off-line

Table 5.3: Elements of cognitive foundations compared

Compilation is usually regarded as a constructive process in the sense that it creates a compiled

[9]+ tests.Kind1.test40IAQualifyingDoubleAccessToNonStaticField()
+ tests.Kind1.test41IAQualifyingDoubleAccessToNonStaticMethod()
+ tests.Kind1.test42IAQualifyingAccessToNonStaticFieldOfStaticInnerClass()
+ tests.Kind1.test43IAQualifyingAccessToChainOfNonStaticFieldsOfStaticInnerClass()
+ tests.Kind1.test44IAQualifyingAccessToNonStaticFieldOfNonStaticInnerClass()
+ tests.Kind1.test45IAQualifyingAccessToNonStaticMethodOfStaticInnerClass()
+ tests.Kind1.test46IAQualifyingAccessToNonStaticMethodOfNonStaticInnerClass()
- tests.Kind1.test47aIAQualifyingAccessToNonStaticFieldOfStaticInnerClassWithAnchorOfEnclosingClass()
- tests.Kind1.test47bIAQualifyingAccessToNonStaticFieldOfNonStaticInnerClassWithAnchorOfEnclosingClass()

binary and the compiler creates an abstract syntax tree (AST) to elaborate the parsed source code structure[10]. 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)[11]. The reliance of compilation on ASTs is the reason that made me choose to implement statically-resolved indirect anaphora over dynamically-resolved ones: anaphora resolution can at compile time be based on existing information in the AST whereas run-time resolution would require analysis of the run-time state of a program containing anaphors. There is also the idea that statically-resolved anaphors are more likely to have referents that have a representation in the source code than dynamically-resolved anaphors and do thus make it easier to understand a piece of source code. This is yet nothing but an unsubstantiated hypothesis, though. Static resolution also fits 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.

5.3.1 Representations of knowledge

In this section I will compare natural language and Java with regard to models of knowledge memorized in the human brain and the involvement of knowledge in understanding texts respectively source code. Before the contribution of knowledge to the TWM can be discussed, models of the representation of knowledge memorized in the human brain need to be considered. For these models, there are two lines of theories in cognitive linguistics: holistic theories assume that all knowledge is stored in a uniform fashion, 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. Schwarz-Friesel's model introduced in chapter 2 is a modular one, but what about Java?

A *lexicon* exists in Java that is involved in the compilation of source files and it is separate from the source files. The lexicon is not explicitly mentioned in the Java language specification which deals with a *lexical grammar* instead (see [GJSB05, 13]) that prescribes how a compiler identifies *tokens* and comments in a stream of characters. Java's tokens I regard as equivalent to words in natural language. Hence, the application of the lexical grammar of Java is equivalent to the identification of words out of single characters that happens when humans look at a text. The Java language specification contains an exhaustive list of keywords, separators and operators and gives production rules for the construction and parsing of valid identifiers and literals (i.e. all forms of tokens) as well as comments (see [GJSB05, 19ff.]) and thus implicitly provides a lexicon for Java. The same may be found in theories of the mental lexicon that may either propose exhaustive lists of words and their forms, rules to recognize and construct them as well

[10]That compilers do, besides reading source code, read libraries in binary form, will not be treated here. I deem libraries that are available in binary form only a special case of code available as source because compilers are equally capable of reading source and binary representations. The same is true for humans, even though binary representations are not made to be read by humans which is why they are not able to read it as efficiently as source code. That is why this case is special – but not entirely different.

[11]One may object that a TWM is a graph, whereas an AST is, by its name, a tree i.e. a restricted form of a graph. This may be true for some compilers, but not for all, though. The JastAddJ compiler used for the implementation described in chapter 7 features node attributes that allow graphs be constructed via attributes attached to the nodes of the AST.

as mixed approaches[12]. Besides this similarity, the lexicon of Java is, unlike the mental lexicon, fixed and its entries are semantically poor because they do not contain any relations between each other as in the case of entries of the mental lexicon (see section 2.4.1). The semantic gap pointed at in section 4.1 thus manifests itself at the lexical level too: not only do the contents of the entries in Java's lexicon and the mental lexicon differ, Java has only a single lexicon entry (a set of production rules) for all identifiers in Java source code, while, from a natural language perspective, there are countless entries in a mental lexicon applying to the different identifiers used in Java source code.

After treating lexical knowledge, conceptual knowledge needs to be looked at. However, I could not identify any conceptual knowledge in Java that (a) exists independent of the source code to be processed by a Java compiler or to be read by a Java programmer and that (b) would contribute to the compilation of Java source code by other means than the fact that the compiler implements the Java language specification. It is at this stage impossible to distinguish modular and holistic models of knowledge in the case of Java due to the lack of text-external knowledge involved in the compilation of Java source code.

If neither lexical nor conceptual knowledge similar to the one involved in human understanding of text is found in Java aside the source code, does Java involve such knowledge at all? This is the case. However, the knowledge is not separated from source code, it is represented within source code. E.g. inheritance hierarchies expressed by *extends* and *implements* clauses in Java express what is known as hyperonymy in linguistics: the relation between a term and its superordinate term. Textual knowledge representation is not only in source code used instead of memorized knowledge, but in natural language as well. It is typical for technical texts to define terms used later. These definitions enumerate the constituents of objects described by a term and how they relate to other objects. I did not read enough on models describing the creation of lexical-semantic and conceptual knowledge in the human brain but I guess that reading representations of knowledge in texts is undoubtedly a way in which mental representations of knowledge can be created and modified – if the information given in the text is memorized instead of being forgotten. Hence, for an optimal (and theoretical) reader who memorizes all knowledge represented in a text, the meaning conveyed by textual and memorized knowledge representation will be identical. Since computer programs are made to be precise, a compiler needs to be the implementation of such an optimal reader. The consequence of this is that in the remainder of this section textual representation of knowledge will be treated as if it was the memorized knowledge representation ad hoc created from that very textual representation of knowledge. The benefit of this is that no form of textual knowledge representation needs to be introduced because the semantically equivalent form of memorized knowledge representation introduced in section 2.4.1 can be used.

Given that knowledge in Java is not separated from source code, could it make sense to separate knowledge from source code? It could be expected that the amount of source code required would be reduced, because the memorized knowledge required to compile the source code would be external to the source code. Just like source code, the memorized knowledge would need to be modular to be re-used in different projects and would need to be transportable so that it could be used by different programmers to compile source code on different computers. The represen-

[12]As mentioned in chapter 2, my knowledge of theories on the structure of the mental lexicon is yet limited.

tation of memorized knowledge would need to be open to examination by programmers because computer programs are typically deterministic and being able to read all information constituting a program is crucial in debugging programs that do not work as expected. A textual representation would fit this purpose well because today's programming systems are made to visualize and manipulate textual representations. The inclusion of memorized knowledge within source code does already fulfill these criteria of modularity, transportability and examinability without requiring programmers to learn another textual representation. Hence, it would not make sense to separate knowledge from source code.

While no actual division between memorized knowledge and texts that represent knowledge is found in Java, it is possible to base the knowledge representation of texts on a modular model that distinguishes lexical-semantic and conceptual knowledge[13]. Based on such a model of knowledge representation in texts, text understanding can be modeled via three-tier semantics and this model can be transferred to the compilation of source code by compilers. This requires that the nodes of the TWM, lexicon entries and conceptual schemata can be identified in the AST, the parsed form of source code processed by a compiler. In the following paragraphs I will identify elements of three-tier semantic in the AST, based on criteria for the three tiers that had been given in sections 2.4.1 and 2.4.2.

As a working hypothesis, I consider the nodes of the AST equivalent to the nodes of a TWM. Theoretically, all nodes of an AST take a special role as a complement of a lexicon entry or (part of) a conceptual schema. This fact reflects what has been stated above: that knowledge in Java is represented within the source code instead of within mental representations external to the source code. This lossless coalescence of textual and memorized representations of knowledge known from natural language during their transfer to Java leads to the dual function of AST nodes[14].

I treat method headers (see [GJSB05, 209f.]) as lexicon entries[15] 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 whose name is used in the class instance creation expression[16]. The *constructor header* is the constructor 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.

Headers of classes and interfaces[17] I treat as if they were lexicon entries of nouns because

[13]It may as well be possible to use a holistic model instead and it may be interesting to seek to demonstrate the equivalence of what holistic and modular models are able to express. I will stick to Schwarz-Friesel's modular model from chapter 2, though.

[14]The coalescence also obviates the need for lexicon entries, concepts and conceptual schema to be created and elaborated in Jaaa. Understanding theories of these processes from cognitive linguistics seems fruitful nonetheless (see future work item 2.6 on page 20).

[15]This work does not deal with the lexical grammar of Java, but with lexicon entries as found in models of the mental lexicon.

[16]Constructors are not invoked directly but a class instance creation expression is used instead that will make the JVM invoke the constructor upon the newly created instance.

[17] By class and interface header I mean the name of a class or interface and all its supertypes. This diverges from the

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[18].

The declarations of fields and accessors[19] 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. The fields and accessors play the role that variables have in frames introduced in section 2.4.1. This choice is obvious since the definition of Stillings et al. reproduced in section 2.4.1 is very close to the definition of a class in Java. 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 model conceptual knowledge (see 2.4.1). Methods, constructors and even initializers 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, constructor or initializer, which is associated to its declaring type and its subtypes and in the case of non-static methods an instance of the declaring type, scripts are not exclusively associated to an entity. Scripts are not investigated as a form of conceptual schemata in this work, they may be considered in future work, though.*

5.3.2 Abstract syntax trees as text-world models

I proposed above that all nodes of the AST are equivalent to the nodes of a TWM as well as to memorized knowledge. This is not completely accurate because bodies[20] are not yet understood as scripts in Jaaa and thus the AST nodes within bodies do not yet represent memorized knowledge but only TWM nodes. The other elements of Java source code outside of bodies I did already describe above as taking the role of memorized knowledge. In this section I will discuss how these AST nodes outside of bodies (hereafter "extra corpus nodes") relate to AST nodes inside bodies that do not stand for memorized knowledge (hereafter "intra corpus nodes").

Lacking an implementation of scripts, *extra corpus* nodes can be understood as representing *given* knowledge – given in the temporal sense that it is available when a method body is to be compiled[21]. Similarly to what had been described in section 2.4.2, text understanding (i.e. understanding of source code within method bodies) can be grouped into two phases[22]: first,

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 in the class or interface header. 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 Jaaa as well.

[18]The other kinds of types known in Java – type parameters and array types – are not treated as part of this work.

[19]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 *accessors* in section 6.3.1 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.

[20]hereafter subsuming bodies of methods, constructors and initializers

[21]When scripts are implemented, the compilation of a method body can depend on the compilation of other method bodies what will lead to more complex dependencies.

[22]I shall be noted that there are theories in cognitive linguistics that assume parallel processing (see [Sch08, 195])

AST nodes are created that reflect the express semantics of the source code (e.g. with references yet unresolved), second, the intra corpus nodes are elaborated with knowledge derived from extra corpus nodes (e.g. while anchoring indirect anaphors). It is at this stage that the broad and the narrow definition of text introduced in section 5.1 overlap. The broad sense of text applies, because most[23] extra corpus nodes are outside of the text defined in the narrow sense. The narrow sense applies, because the anchor of an indirect anaphor can only be within the same body - i.e. within the same text in the narrow sense.

It is already true for existing features of the Java programming language that intra corpus nodes relate to extra corpus nodes, e.g.

- Simple expression names (see [GJSB05, 134]) relate to the declaration of a local variable, a parameter or a field (of which only the latter is extra corpus).
- A field access expression relates to a field declaration,
- the same is true for method invocation expressions and method declarations
- as well as for class instance creation expressions and class declarations.
- Additionally, the type of the result of an expression is connected to a class or interface declaration.

Analogously, indirect anaphors relate to extra corpus nodes in order to be anchored, and, like expressions accessing local variables or parameters also incorporate intra corpus nodes in referentialization. E.g. implementing meronymy-based anchoring (see section 2.5.2) in Java requires an (intra corpus) anchor that is related to an (extra corpus) class declaration that declares a field for which a meronymic indirect anaphor stands.

The two phases of text understanding already identified in compilation as well may be called parsing and transformation stages. During the parsing stage, instantiation of nodes takes place through which the parser constructs a syntax tree that represents what is directly expressed in the parsed source code. This syntax tree can already be abstract in the sense that brackets are excluded because precedence can be modeled by the tree's structure instead. The tree created by the parser is further abstracted in a second stage through transformations that correspond to the elaboration phase in text understanding. In the case of abovementioned existing features of Java, the nodes created by the parser are related to nodes representing declarations and the relation is established based on identical identifiers used in the parsed node and the declaration as well as based on contextual information. To anchor indirect anaphors instead, it is necessary to find a suitable anchor among the intra corpus nodes that has a header or frame the indirect anaphor can be anchored in. Different forms of anchoring and the nodes involved will be described below. A final important point is to be made on defaults.

In chapter 2 it was put forward that conceptual schemata provide defaults that indirect anaphors can replace. It was said that during text understanding, for each occurence of an indirect anaphor

i.e. processing of the phases may not be strictly sequential as long as there are no dependencies between them that force sequential processing. Section 5.3.4 below will discuss this issue further.

[23]Note that headers of method or constructor declarations are extra corpus, but still belong to the text in the narrow sense because the headers of methods and constructors are part of the corresponding anaphoric scopes.

a new nodes is created in the TWM that is used instead of the default provided by the conceptual schema that the new node replaces. To understand what that means during the compilation of a Jaaa program, the following questions need to be answered. What is a default in Java? Which kind of nodes represent indirect anaphors in Java? Which kinds of nodes are the ones that are instantiated to replace defaults? How does the replacement take place? The answer to the first question will be developed in the subsequent section 5.3.3: the declared types of fields and the return and argument types of accessors do in Java serve the same purpose as defaults do in text understanding. Details on the kinds of nodes representing indirect anaphors will be deferred until chapter 6. I will not yet attempt a generalized discussion of the other questions but defer it to a later version instead, when further kinds of indirect anaphors have been implemented.

5.3.3 Defaults, initializers and declared types

Conceptual schemata contain variables for concepts. These variables can have a default value. Do default values exist in Java as well? It was above established that fields of classes equal variables in frames. Field declarations can have an initializer, which is, however not identical to a default value of a variable in a frame. While initializers can depend on or trigger arbitrarily complex computation at run-time and it is not expected that the resulting value can be inferred reliably at compile-time, default values are available during text understanding (the equivalent of being available at compile-time). I.e. it looks as if there are no defaults in Java. What is the role that defaults play while understanding texts in natural language?

Schwarz-Friesel gives examples that involve defaults, the two of which that are relevant for treating the function of defaults I reproduce here (translations mine). (1) "Martin bekam ein Buch" [Sch08, 118] ("Martin got a book"). While reading this sentence, a default GIVER is activated, even though the GIVER is never referred to - i.e. text understanding would, as far as I can see, have worked equally well, hadn't the default been activated. (2) "Jürgen besuchte ein Restaurant in Tunis. Der Kellner erhielt ein großzügiges Trinkgeld." [Sch08, 119] ("Jürgen ate out in Tunis. The waiter received a generous tip."). In this case, the default value for WAITER in a RESTAURANT frame or script previously activated by the phase "ate out" can be said to be replaced while reading by a newly instantiated WAITER node constructed for "The waiter". I.e. the reading of both sentences leads to an understanding without using the default value in the TWM. Even if one considers that a RESTAURANT script is acted out by a person, I could not yet imagine the default for waiter being useful except for cases when an actual waiter is present whose representation would then replace the default in the mental model. Since I cannot yet find a use of the value of a default, but defaults are clearly used to identify matching replacement values, I consider default values known from text understanding to be equivalent to the declared types of fields and the parameter and return types of accessors in Java programming languages, but not to values, initial values or default values known in programming languages.

Future Work 5.2 (Functions of default values of schemata) *The literature may provide information on when default values may actually function as more than means to identify suitable replacement values and as targets to replacement.*

5.3.4 When elaboration happens

In section 2.4.2 I highlighted that (in theory) text understanding is a two-step process in which an initially created TWM is elaborated in a later step and that the two steps may be parallelized. Parallelization is the norm, but not mandatory: if all required information is available when an textual entity is read, construction and elaboration of the corresponding TWM nodes will be immediate or *on-line* (see [Sch08, 194f.]). If information is lacking, construction of the TWM nodes happens immediately but elaboration is deferred until the reader reaches a later point in the text that provides the information required for so-called *off-line* elaboration. Two-phase interpretation was also said to happen when a compiler works on the AST (see section 5.3.2). However, I guess that all compilers create a syntax tree that immediately represents the parsed source code before beginning compilation i.e. elaboration. I.e. it is not possible to elaborate a node of the syntax tree on-line i.e. just at the moment when it has been created[24].

5.4 Semantics

Element	Natural Language	Jaaa
Thematic roles	AGENT, PATIENT, THEME ...	Argument- and return types of methods
Nominal relations	Hyperonymy, meronymy ...	Supertypes, fields and accessors
Referent of Anchor	Node in TWM	Node in AST related to the header of: method, class or interface
Referent of IA	Node in TWM	Node in AST related to the header of: class or interface
Anchoring	As per section 2.5.5	Analogous to natural language, theme yet unsupported
Referential ambiguity	Possible	Reducible

Table 5.4: Elements of semantics compared

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

[24]This may be an over-generalization, but it is at least true for the JastAddJ compiler that the implementation from chapter 7 is based on and I will stick to this potential over-generalization as a working hypothesis.

normally are no synonymy relations in the type graphs used within Java programs because one typically strives to avoid redundancy in source code.

5.4.1 Indirect anaphors

Syntactically, an indirect anaphor is a primary expression in source code (see section 5.2). Its semantics are defined as follows. The anaphoric scope that the indirect anaphor occurs in is called the anaphoric scope of the indirect anaphor. When successfully anchored, an indirect anaphor relates to another source code entity in its anaphoric scope. This source code entity is said to function as the *anchor* of the indirect anaphor. The relation between indirect anaphor and anchor is called *indirect anaphora*. (Note that it is possible that more than one indirect anaphor is anchored in an anchor[25].) An anchor must have a type and thus relate to a lexicon entry that is either (a) the header of a class or of an interface or (b) the header of an invocable. Furthermore, in case (a) the lexicon entry is related to a conceptual scope which is a frame. The header (and, if available, the frame) constitute the cognitive domain of the anchor. Both the header and, if available, the frame of an anchor are used to anchor indirect anaphors at compile-time. They manifest in the AST in the form of nodes representing the declaration of an invocable, class or interface. The indirect anaphor has a type, too (the type of the indirect anaphor), which is mentioned in its source representation and leads to the indirect anaphor's node in the AST being related to a class or interface declaration which is involved in anchoring as well[26].

The anchor source-code entity stands in a reference relation to a node in the AST which serves as its referent. The same is true for the indirect anaphor. After parsing, the referent of an indirect anaphor will typically be a node called *indirect anaphor*. This node will typically be transformed into a node of another type during compilation. Possible referents after transformation are specific to the kind of indirect anaphor. Note that the reference relation is purely notional – i.e. not manifest in the AST or any other structure.

Besides the mentioned compile-time semantics, expressions have run-time semantics as well concerning which the Java language specification defines the following.

[25]- tests.General.test50Kind1IANotAnchoredForLackOfAnchors()
+ tests.General.test51Kind1IAAnchoredInSingleAnchor()
+ tests.General.test52TwoKind1IAsAnchoredInOneAnchor()
- tests.General.test53ThreeKind1IAsAnchoredInOneAnchor()

[26]+ tests.General.test10IAWithTypeInteger()
- tests.General.test11NoPrimitiveIA()
+ tests.General.test12IAWithTypeJavaLangInteger()
+ tests.General.test13IAWithTypeJavaUtilVector()
+ tests.General.test14IAWithImportedTypeVector()
+ tests.General.test15IAWithImportedTypeVectorWithTypeParameterString()
+ tests.General.test16IAWithTypeJavaUtilVectorWithTypeParameterString()
+ tests.General.test17aIAWithTypeOfSameClass()
+ tests.General.test17bIAWithTypeOfEnclosingClass()
+ tests.General.test17cIAWithTypeOfInnerClass()
+ tests.General.test17dIAWithTypeDefinedInSameFile()
- tests.General.test18IAWithArrayType()
+ tests.General.test19aIAWithParameterizedType()
- tests.General.test19bIAWithUnknownType()

> When an expression in a program is *evaluated* (*executed*), the *result* denotes one of three things:
>
> - A variable (§4.12) (in C, this would be called an *lvalue*)
> - A value (§4.2, §4.3)
> - Nothing (the expression is said to be void)
>
> [GJSB05, 409], emphasis in original

Applied to indirect anaphors, this means that they can at run-time be evaluated to a result. In the case of indirect anaphors, a result denotes either a value or a variable. I will not distinguish between the result and its denotatum but will instead say that an indirect anaphor has or denotes a result which is a value or a variable. The result of the indirect anaphor is specific to the kind of indirect anaphora. (The specifics of the different kinds of indirect anaphora in Jaaa are given in chapter 6.) The phrase *the value of an indirect anaphor* covers the value of an indirect anaphor that denotes a value as well as the value of a variable of an indirect anaphors that denotes a variable (cf. [GJSB05, 410]). The type of the result of an indirect anaphor is assignment compatible (see [GJSB05, 95]) with the type of the indirect anaphor, unless heap pollution occurs (cf. [GJSB05, 410])[27]. Assignment compatibility is, except in the case of heap pollution, ensured at compile time. An indirect anaphor can not yet denote a value of a primitive type. This design choice is motivated by the fact that primitive types have no header or frame. It may be waived later. Note that the result of an indirect anaphor is a run-time entity and thus not represented in the AST.

5.4.2 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 anaphoric scope that contains the indirect anaphor, i.e. identify a souce code entity whose cognitive domain has a place for the referent of the indirect anaphor[28]. A suitable anchor is determined by applying the following conditions[29] that filter the set of potential anchors.

[27]+ tests.General.test40IAWithDirectSuperclassOfAnchorType()
+ tests.General.test41IAWithIndirectSuperclassOfAnchorType()
+ tests.General.test42IAWithInterfaceImplementedByAnchorType()
+ tests.General.test43IAWithInterfaceImplementedBySuperclassOfAnchorType()
- tests.General.test44IAWithSubclassOfAnchorType()
- tests.General.test45IAWithSiblingOfAnchorType()

[28]See chapters 6 and 7 for specifics.

[29]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 need to be adapted.

a) The anchor must occur within the anaphoric scope of the indirect anaphor and be identical to or part of a statement before[30] the statement that contains the indirect anaphor[31].

b) No void: If the anchor is a method invocation expression, the method invoked must not return void[32].

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[33].

d) No literals: Literal expressions cannot be used as anchors[34,35].

e) No arguments of explicit constructor invocations: Explicit constructor invocations that can appear as the first statement in a constructor body can have arguments. These arguments cannot function as anchors[36,37].

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[38]. Note that multiple occurrences of an expression in the source code can lead to referential ambiguity (see section 5.4.3 for a discussion) if the occurrences denote the same variable.

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 of the anchor 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 cognitive domain of the potential anchor must be part of the currently focused theme. As has been explained in future work item 2.10, I did not find a good model of theme. Frames, the only form of conceptual schemata yet transferred to Jaaa, cannot be used as themes directly because they are relatively find-grained while Schwarz-Friesel also used rather coarse-grained scripts to establish a theme what looks more promising but is not possible here, since scripts have not yet been transferred. Jaaa will thus not use themes right so far. Focus and activity will thus be irrelevant as well.

[30]- tests.General.test07NotCataphoric()

[31]This is choice made to avoid cases like the right-hand operand of an assignment being an indirect anaphor that denotes the value of the left-hand operand of the assignment. If it is later found that such constructs are not irritating, they may be implemented.

[32]- tests.General.test03NoVoidMethodAccess()

[33]- tests.General.test04NoPrimitiveAnchor()

[34] This is an arbitrary decision that may be waived later, when primitives can be used as anchors.

[35]- tests.General.test02NoLiterals()

[36]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. This limitation may be waived later.

[37]- tests.General.test05NoArgumentOfExplicitConstructorInvocationAsAnchor()

[38]- tests.General.test20ReferentialUnambiguityWRTReturnTypes()

   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. [39]

2. Establish the relation between indirect anaphor and the suitable anchor in one of the following ways.

   a) If the anchor relates 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 6.2.

   b) Otherwise, if the anchor relates 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 frame contains an accessor method that is compatible with the type of the indirect anaphor, indirect anaphora will be created as described in section 6.3.

   c) If the anchor relates to the header of a class or interface whose frame is not suitable for anchoring according to 2a and 2b, indirect anaphora will be inferred as described in section 6.4.

5.4.3 Referential ambiguity

Referential ambiguity means that a single suitable anchor provides access to more than one suitable referent of the indirect anaphor or that there are at least two suitable anchors available leading to more than one suitable referent of the indirect anaphor as well. 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[40].

5.5 Naturalness

The transfer of indirect anaphora from natural language to programming languages in the form of a metaphor is supposed to make programming languages more natural (i.e. known or easy

[39]- tests.General.test30IgnoreExpressionsInSameStatement()
- tests.General.test31IgnoreExpressionInThirdLastVariableDeclaration()
+ tests.General.test32AnchorToExpressionInSecondLastStatement()

[40]+ tests.Kind2.test10ReferentialAmbiguity()
+ tests.Kind2.test11PreventedReferentialAmbiguityWhenAnchoringInLocalVariableDeclarationWithInitializer()

to learn, cf. section 3.2). It was in that section demanded that for features that are supposed to be natural it should be argued why they are supposed to be natural. What does this mean in the case of the metaphor of indirect anaphora added to programming language? Why is it supposed to be natural? Because it is rooted in cognitive theories of natural language. If these theories accurately describe natural language, and if their implementation in programming languages matches the theories, then indirect anaphors will be known or easy to learn. Up to the proof of this hypothesis by experiments involving human programmers, this chapter will remain preparatory scaffolding for the proof.

5.6 Summary

This chapter combined findings from chapters 2, 3 and 4 to outline a metaphor of indirect anaphora transferred from natural language to Java. At the level of pragmatics, the sovereignty of definition of the compiler and reading order were highlighted, before a broad and a narrow definition of text were developed. Syntactically, structures including sentences and phrases were discussed and the syntax of indirect anaphors in Jaaa has been described. Concerning cognitive foundations it was found that Schwarz-Friesel's modular semantics can be mapped to Jaaa even though textual and memorized knowledge is identical in Jaaa. Nodes in the AST of Jaaa are equivalent to nodes of the TWM and extra corpus nodes are equivalent to memorized knowledge, because an interpretation of bodies of methods and constructors as scripts has not yet been implemented. The elaboration of the AST was discussed as well and it was concluded that it typically happens off-line. Elaboration is based on lexicon entries that in Jaaa are not identical to information provided to Java's lexical grammar, but to headers of invocables (i.e. parameter and return types) or headers of classes and interfaces (i.e. direct and indirect supertypes). Elaboration also makes use of frames that in Jaaa means the fields, accessors, direct and indirect supertypes of a type. It turned out that defaults in conceptual schemata do not seem to be identical to initializers in Java and need further study. Concerning semantics it was fixed that thematic roles are reflected in declared return types and types of parameters, that hyperonymy and meronymy are manifest in the declaration of supertypes, fields and accessors. The referent of an anchor was said to be a node in the AST that is related to the header of a method, a class or an interface. The referent of an indirect anaphor was said to be a node in the AST related to the header of a class or interface. Anchoring in Jaaa is analogous to what had been described in section 2.5.5 except that no thematic progression is implemented, partly because solitary frames identified in Java seem to be too fine-grained to be used as theme. It was found that referential ambiguity can in some cases be automatically reduced in Jaaa. The chapter closes by contextualizing the metaphor of indirect anaphora as added to programming languages: it is pointed out that the roots of the constructed metaphor are in cognitive linguistics, that these roots are suppposed to make the metaphor natural – i.e. known or easy to learn – and that the chapter is only preparatory scaffolding for a future proof of this hypothesis.

6 Indirect Anaphora for Java

Having mapped entities found in models used in cognitive linguistics to the Java programming language, I will now use the mapped entities to transfer specific kinds of indirect anaphora from section 2.5 to Jaaa. This transfer will abstract the specifics of an implementation of the transferred kinds of indirect anaphora, which will be detailed in chapter 7. Because this chapter abstracts implementation details, it will use terminology from the Java language specification as far as possible. Each kind of indirect anaphora will be introduced by an example as well as pre- and postconditions that need to be met to comply to the kind of indirect anaphora. Before specific forms are introduced, pre- and post conditions that apply to all kinds of indirect anaphora are given.

6.1 General Properties of Indirect Anaphors in Java

All occurences of indirect anaphors must satisfy the following conditions.

6.1.1 Preconditions

For an indirect anaphor to be compilable, the following conditions must hold.

1. The anchor is suitable, i.e. it satisfies conditions 1a - 1f, 1h, 1i given on pages 46f.
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]).

6.1.2 Postconditions

After an indirect anaphor has been compiled, the following conditions must hold.

1. Run-time evaluation of the indirect anaphor must not be affected by concurrent computation happening between the execution of the anchor and the evaluation of the indirect anaphor.

6.1.3 Invariants

Before, during and after the compilation of an indirect anaphor, the following conditions must hold.

1. If the anchor is an expression, the number of times that it is evaluated is the same as when no indirect anaphor was in an indirect anaphora relation to it.

2. Line numbers are the same as before compilation of the indirect anaphor[1].

6.2 Kind 1: Anchoring Based on Headers of Invocables

The simplest form of indirect anaphora in Jaaa occurs when an indirect anaphor relates to a method invocation expression or a class instance create expression and denotes the value returned or created by the expression. This form is analogous to anchoring based on thematic roles described in section 2.5.1. After an example, the pre- and postconditions of this anchoring strategy are given below in an implementation-indepedenten form. The details of an implementation will be elaborated in section 7.2.

6.2.1 Example

Consider listing 6.1 below that shows a method that is part of the JUnit code base[2]. In the method, a `Result` instance is returned by a method invocation, stored in a local variable `result` and accessed immediately afterwards.

```
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 6.1: Snippet from org.junit.runner.JUnitCore (JUnit 4.8.2)

The variable `result` is replaced by use of an indirect anaphor `.Result` in listing 6.2 that was compiled and passed the JUnit tests of JUnit afterwards.

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

Listing 6.2: The indirect anaphor `.Result` in a modified version of listing 6.1

The indirect anaphor `.Result` is anchored in the method invocation in line 53 and thus does at runtime denote the value returned by the method invocation, replacing the previously used local variable. (Implementation details on this examples are deferred until section 7.2.1.) In the sample, the indirect anaphor shortens the source code by hiding where the result comes from. I expect that programmers who are not familiar with the internals of JUnit but know the semantics of indirect anaphors will be able to guess that it is obtained from the invocation of `runMain`. Whether this is true in general needs to be shown by experimental evidence, though.

[1]+ tests.Kind1.test60LineNumbersInSubsequentExceptionsCorrespondToSourceCode()
+ tests.Kind1.test61LineNumbersInExceptionIncludingAnchorInStackTrace()

[2]version 4.8.2 was used, available from https://github.com/downloads/KentBeck/junit/junit-4.8.2-src.jar

6.2.2 Preconditions

For an indirect anaphor of kind 1 to be compilable, the following conditions must hold.

1. The general preconditions laid out in section 6.1.1 are met.
2. The anchor is a method invocation expression or a class instance creation expression[3].
3. The anchor refers to the header of an invocable that is accessible[4].
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].

A compile-time error is raised if any of the abovementioned conditions is not met.

6.2.3 Postconditions

After an indirect anaphor of kind 1 has been compiled, the following conditions must hold.

1. If the anchor is a method invocation expression, the indirect anaphor does at runtime denote the value that has been returned by the method invocation expression when the latter was evaluated.

[3]+ tests.Kind1.test01AnchorInMethodAccess()
+ tests.Kind1.test02AnchorInMethodAccessWithinDot()
+ tests.Kind1.test03AnchorInMethodAccessInitializingAVariable()
+ tests.Kind1.test04AnchorInMethodAccessUponTheResultOfAMethodAccess()
+ tests.Kind1.test05AnchorInArgumentOfMethodAccess()
+ tests.Kind1.test20AnchorInClassInstanceExpr()
+ tests.Kind1.test21AnchorInClassInstanceExprWithinDot()
+ tests.Kind1.test22AnchorInClassInstanceExprInitializingAVariable()

[4]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 during the compilation of the method invocation expression or class instance creation expressions.

[5]- tests.Kind1.test90NoIAOnLeftHandSideOfAssignment()
+ tests.Kind1.test91IARightHandSideOperandOfAssignment()
+ tests.Kind1.test92IAInitialValueOfVariableDeclaration()
+ tests.Kind1.test93IAInitialValueOfVarDecl()
+ tests.Kind1.test94IAPartOfFieldAccessOnLeftHandSideOfAssignment()
+ tests.Kind1.test95IAPartOfMethodAccessOnLeftHandSideOfAssignment()
+ tests.Kind1.test96IAAsArgumentInMethodAccess()

2. If the anchor is a class instance creation expression, the indirect anaphor does at runtime denote the value that has been returned by the class instance creation expression when the latter was evaluated.

3. The general postconditions laid out in section 6.1.2 are met.

A compiler does not provide a valid implementation of anchoring based on the header of invocables if any of the abovementioned conditions is not met.

6.2.4 Invariants

Before, during and after the compilation of an indirect anaphor of kind 1, the following conditions must hold.

1. The general invariants laid out in section 6.1.3 are met.

Future Work 6.1 (Underspecification of arguments) *The current implementation of anchoring based on headers of invocables is only able to denote 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 are 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. It is obvious that such a feature would need a syntactically special form of supplying arguments to methods so no problems arise in the case of overloaded methods that can have multiple variants of a method with fewer arguments, just like in the case of underspecification.*

6.3 Kind 2: Anchoring Based on Fields and Accessors

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 and allows parts of instances to be accessed[6]. The remainder of this section are devoted to definitions, an example, pre- and postconditions. The details of an implementation will be elaborated in section 7.3.

Note that the description of this kind of anaphor is mostly a sketch that has not yet been implemented and may need to be altered during implementation.

[6]In footnote footnote 17 on page 40 I explained that headers to not contain meronymic information but only frames do and that information on mandatory parts and otherwise directly related concepts are not distinguished in Jaaa. Thus, the transferred kind of indirect anaphora integrates two kinds of indirect anaphora described by Schwarz-Friesel who distinguished meronymy-based anchoring based on mandatory parts mentioned in the lexicon entry (see section 2.5.2) and schema-based anchoring based on directly related concepts mentioned in conceptual schemata (see section 2.5.3).

6.3.1 Accessors

Accessors are used to enable other objects to access inaccessible 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.

6.3.2 Example

Listing 6.3 shows code from JHotDraw version 7.0.8[7] that is at the beginning of a method body and uses none of the arguments supplied to the method upon invocation. In the snippet, there are a number of method invocations on the object stored in the `view` field.

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

Listing 6.3: Snippet from org.jhotdraw.draw.BezierControlPointHandle.java (JHotDraw 7.0.8)

In listing 6.4 one of the method invocations is replaced by an indirect anaphor of kind 2. Note that this listing has not been compiled or tested, because the depicted variant of indirect anaphora of kind 2 has not yet been implemented.

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

Listing 6.4: The indirect anaphor `.Constrainer` in a modified version of listing 6.3

It may seem strange that no indirect anaphor is used at line 83 already to replace the invocation of `view.getDrawing()` but `view.getConstrainer()` in line 84 is replaced instead.

[7]http://sf.net/projects/jhotdraw/files/JHotDraw/JHotDraw%207.0.x/jhotdraw-7.0.8.nested.zip/download

The reason for this is that the anchor of `.Constrainer` in line 84 is the access to `view` in line 83. The invocation of `view.getDrawing()` in line 83 cannot be replaced because there is no prior access to `view` in the method body i.e. no anchor is available.

I expect this sample to be compilable once the implementation allows for invoking getter methods on an anchor of an indirect anaphor.

Similar to the example in section 6.2.1, above use of indirect anaphora shortens the source code by hiding where the `Constrainer` instance is obtained from. Whether this is going to irritate or not needs to be shown in experiments measuring the fraction of programmers who are able to understand the code (see section 8.2).

6.3.3 Preconditions

For an indirect anaphor of kind 2 to be compilable, the following conditions must hold.

1. The general preconditions laid out in section 6.1.1 are met.

2. The anchor fits one of the following cases.

   a) It is a parameter declaration or a local variable declaration[8]. (The type referenced in the declaration of the parameter or local variable is hereafter called *the type of the anchor*. The value that the parameter holds at run-time is hereafter called *the value of the anchor*.)

   b) It is an access to an accessible field, an access to a local variable, an access to a parameter, another indirect anaphor, or any other expression. (The type of the expression is hereafter called *the type of the anchor*. The value to which the expression is evaluated to at run-time is hereafter called *the value of the anchor*.)

3. If the indirect anaphor is not the left-hand side operand of an assignment expression it denotes either a variable or a value and only a single one of the following two cases applies.

   a) The type of the anchor declares or inherits exactly one field[9] that is visible, accessible (and static, if the indirect anaphor appears in a static method or in a static initializer) and whose type is assignment compatible with the type of the indirect anaphor.

   b) The type of the anchor declares or inherits exactly one getter method for which the following is true at the location of the indirect anaphor: the getter method 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.

[8] + tests.Kind2.test02ReadPublicFieldFromVariableDeclaration()
[9] - tests.Kind2.test01ReadNonExistentField()

4. If the indirect anaphor is the left-hand side operand of an assignment expression[10] it denotes a variable[11] and only a single one of the following two cases applies.
    a) The type of the anchor declares or inherits exactly one field for which the following is true at the location of the indirect anaphor: the field is visible, accessible, non-final (and static, if the indirect anaphor appears in a static method or in a static initializer) and the type of the indirect anaphor is assignment compatible with the type of the field.
    b) The type of the anchor declares or inherits exactly one setter method for which the following is true at the location of the indirect anaphor: the setter method 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.

A compile-time error is raised if any of the abovementioned conditions is not met.

6.3.4 Postconditions

After an indirect anaphor of kind 2 has been compiled, the following conditions must hold.

1. If precondition 3a applies, the indirect anaphor does at runtime denote a variable, namely the field indicated by precondition 3a which is provided by the object that is referred to (in Java terms) by the value of the anchor. The field is accessed once every time that the indirect anaphor is accessed, its value is not cached[12]. The value of the indirect anaphor will be null if the value of the field is null.

2. If precondition 3b applies, the indirect anaphor does at runtime denote a value, namely the value returned by an invocation of the getter method on the object that is referred to (in Java terms) by the value of the anchor. The getter method is accessed once every time that the indirect anaphor is accessed. Its value is not cached[13].

3. If precondition 4a applies, the assignment expression mentioned in the precondition will at run-time assign the value that its right-hand side has been evaluated to to the variable denoted by the indirect anaphor. The variable denoted by the indirect anaphor is the field indicated by precondition 4a which is provided by the object that is referred to (in Java terms) by the value of the anchor. The field is assigned once every time that the indirect anaphor is assigned a value.

[10]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.

[11]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.

[12]This is required so concurrent modifications of the field's value are reflected in the value of the indirect anaphor.

[13]for the same reason as in the previous postcondition

4. If precondition 4b applies, the indirect anaphor does not have a result at runtime. Instead, the assignment expression mentioned in the precondition will at run-time pass the value that its right-hand side has been evaluated to as an argument to the setter method indicated by precondition 4b. The setter method is invoked on the object that is referred to (in Java terms) by the value of the anchor. The setter method is invoked once every time that the indirect anaphor is assigned a value.

5. A NullPointerException or a subclass of it will be thrown at runtime if the value of the anchor is null. The exception shall have a message stating a broken indirect anaphora relation and indicate the anchor whose value could not be obtained, the first indirect anaphor that refers to the anchor, the line number at which the first indirect anaphor occurs and the number of indirect anaphors referring to the anchor if there is more than one of them. The exception shall be thrown at the line of code in which the anchor occurs or begins.

6. The general postconditions laid out in section 6.1.2 are met.

A compiler does not provide a valid implementation of anchoring based on the header of invocabled if any of the abovementioned conditions is not met.

6.3.5 Invariants

Before, during and after the compilation of an indirect anaphor of kind 2, the following conditions must hold.

1. The general invariants laid out in section 6.1.3 are met.

6.4 Kind 3: Inference-Based Anchoring

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

6.5 Summary

This chapter detailed descriptions of different kinds of indirect anaphors in Jaaa using pre- and postconditions and invariants. All kinds share a number of conditions that need to be met, that deal with concurrency, execution of anchors and line numbers. Three kinds have been proposed. The first one, which has already been implemented, was derived from anchoring based on thematic roles and accesses the value returned or created by a method invocation expression or class instance creation expression. The second kind has been implemented in minor parts only. It is derived from meronymy-based and schema-based anchoring at the same time because Java does not allow for a distinction between parts of an instance and objects directly related to it. Indirect anaphors of the second kind can appear as the left-hand side operand of assignment expressions which makes them function as variable assignment or setter invocation. In all other positions, they will function as variable access or getter invocation. An annotation may be introduced that marks unconventional accessors. The third kind of indirect anaphors, to be modeled after inference-based anchoring has not yet been specified but will be based on traversing multiple invocations of accessors and is expected to require an implementation of thematic progression.

7 Implementation

The implementation of Jaaa[1] is based on the JastAddJ compiler (see [EH07]) which is a modular Java compiler based on the JastAdd compiler construction system[2] (see [EH06]). JastAdd uses an aspect-oriented derivative of Java similar to AspectJ in order to allow compilers to be programmed. Features of JastAdd that proved useful for my work include attributes and rewrites. JastAddJ uses a JFlex[3] lexical grammar that I did not change in the course of my work. The compiler is based on a parser created by the Beaver LALR(1) parser generator[4] the grammar of which I had to modify in order to implement indirect anaphora in Jaaa. In this chapter I will provide details on the implementation of the different kinds of indirect anaphors specified in the previous chapter as well as an overview over the test cases used to drive development. First of all, the extent of the limitations of the implementation is illustrated, though.

7.1 Limitations of the Prototype

To give an idea of the limits of my prototypical implementation, the following non-exhaustive list gives some shortcomings of the compiler that I am aware of. I am aware of further shortcomings that are not in the list and there will be limitations that I am not aware of. It can generally not be expected that the compiler is able to fulfill tasks not covered by the test cases (see the blue footnotes throughout this document as well as table 7.1 on page 65).

- Exceptions declared by accessors are ignored by the prototype.
- The type of expressions that act as a potential anchor cannot be an array type or type variable
- The type used in the IA cannot be an array type or type variable, generics/parameterized types are not fully supported
- If more than one indirect anaphor is anchored in an anchor, multiple local variables are created instead of sharing one.
- Indirect anaphors cannot be anchored in parts of ForInit and ForUpdate expressions or parameters of methods or constructors.
- The compiler was not yet tested against real-world source code.
- Errors unrelated to indirect anaphora can lead to error messages related to indirect anaphora.

[1]See http://monochromata.de/shapingIA/.
[2]See http://jastadd.org/
[3]See http://jflex.de/
[4]See http://beaver.sourceforge.net/

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

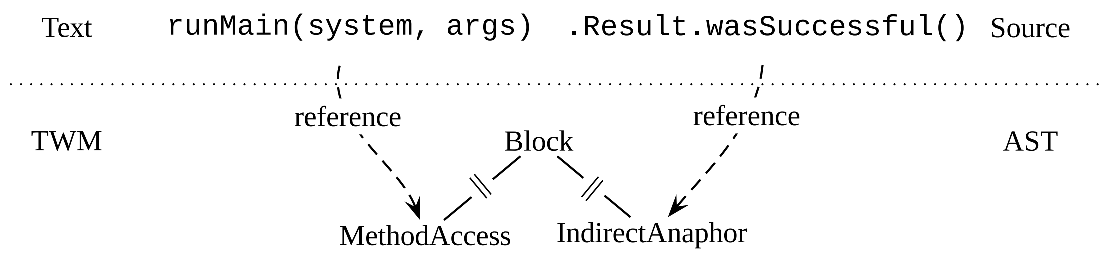

Figure 7.1: Listing 6.2 reproduced, accompanied by the AST created by the parser and reference relations

7.2 Kind 1: Anchoring Based on the Headers of Invocables

This section describes an implementation of what has been specified in section 6.2. Therefore, the example given in section 6.2.1 is elaborated to provide insights into the implementation and an anchoring algorithm is given that transforms an AST that fulfills the preconditions given in section 6.2.2 into an AST that fits the postconditions given in section 6.2.3.

7.2.1 Example

While section 6.2.1 discussed two listings showing a code snippet before and after replacing a local variable by an indirect anaphor, in this section I will discuss the compilation of the snippet in the second listing 6.2, directly after parsing and after the AST has been transformed.

Figure 7.1 repeats listing 6.2 and provides a simplified AST as generated by the parser. To keep the diagram compact, the indirect anaphora relation between the anchor `runMain(system, args)` and the indirect anaphor `.Result.wasSuccessful()` and the lexicon entry related to the anchor are not included in the diagram. The AST depicted also omits a number of nodes (e.g. the MethodAccess node is actually not a direct child of the Block) – these omissions are signalled by the crossed edges in the AST.

Yet the diagram is made to highlight resemblances to indirect anaphors anchored based on thematic roles as illustrated in figure 2.2 on page 16. At the sides of figures 7.1 and 2.2 text has been identified with source code and this identification is in line with the broad definition of text from section 5.1. TWM and AST have been identified based on section 5.3.2. Along these lines, the AST nodes act as referents of the anchor and the indirect anaphor. There is no relation between the MethodAccess and the IndirectAnaphor node yet, since at the end of parsing text

```
52 public static void runMainAndExit(JUnitSystem system,
       String... args) {
53     Result $anchor$0 = null;
54     new JUnitCore().runMain(((Result)($anchor$0 = system)),
         args);
55     system.exit($anchor$0.wasSuccessful() ? 0 : 1);
56 }
```

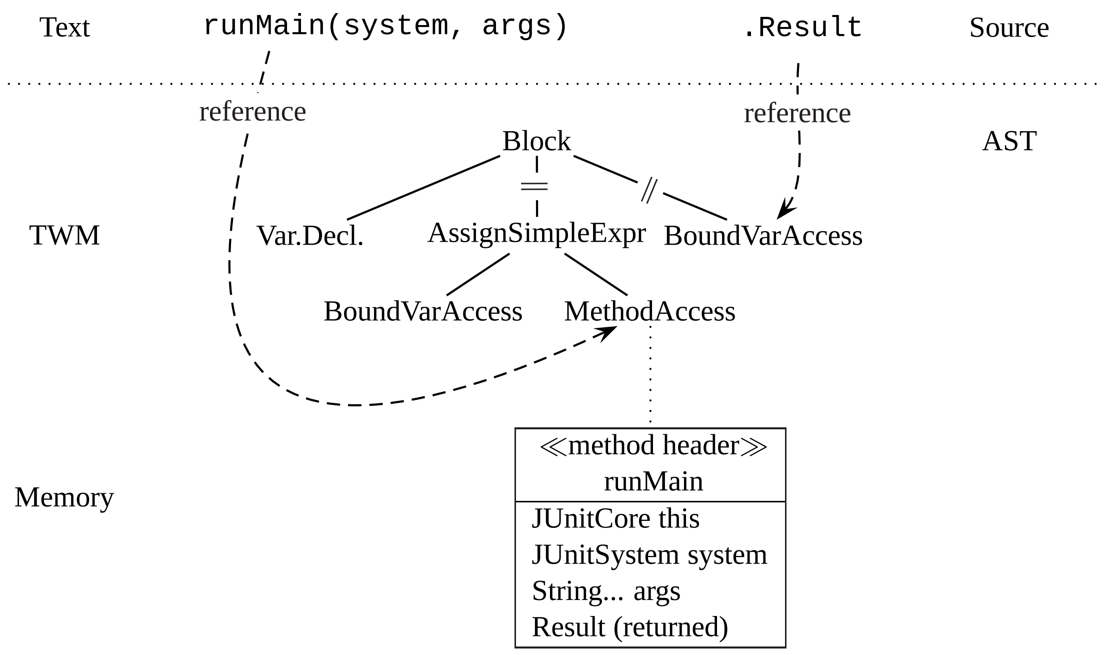

Figure 7.2: Compiler transformations applied to contents of figure 7.1

understanding is not yet complete. The IndirectAnaphor node stems not only from the source entity `.Result` but from the entire fragment `.Result.wasSuccessful()` because syntax does not allow the parser to determine the end of the indirect anaphor in such an expression which is why the entire expression is used to form an IndirectAnaphor node that is split up later on.

Figure 7.2 shows selected elements and relations of the source code and AST created by the compiler for listing 6.2 at a state when all transformations possible to the AST have been performed. The listing at the top of the diagram shows how source code would look after such a transformation (were the transformation applied to the source as well and not only to the AST). This diagram is again simplified: only types of AST nodes are given, not the individual attributes they contain and the crossed edges between Block, AssignSimpleExpr and BoundVarAccess mask the complexity of the expressions in lines 54 and 55 of the listing – actually, the children of the Block are all from different lines of the listing.

The source code fragments of the anchor and indirect anaphor that the dashed reference relations in the diagram originate from are not identical to the fragments in figure 7.1: the IndirectAnaphor node has been split up and its remainder has been rewritten to a BoundVarAccess node during the transformation. This BoundVarAccess, just like the other BoundVarAccess, refers to the VariableDeclaration (Var.Decl.). All of these nodes have, along with the AssignSimpleExpr (an assignment expression) been introduced by the transformation, as can be seen by comparing the listing in the figure to that of figure 7.1.

Compared to the TWM in figure 2.2 the AST of an indirect anaphor in Jaaa is way more complex, especially with regard to the relation between the referents of indirect anaphor and anchor: in figure 2.2 the relation is simply an edge between the nodes of the referents, while in figure 7.2 only an indirect relation exists that traverses a number of edges. This is due to my limited knowledge of the models used in cognitive linguistics and their theoretical, static nature whereas Jaaa is a real implementation that incorporates dynamics such as data flow that complicate the AST. A similarity between natural language and Jaaa is the use of the anchor's lexicon entry (here: method header) in order to anchor the indirect anaphor during AST transformation. Instead of defaults as in figure 2.2, the method header contains types (see section 5.3.3) and even an argument name that is currently unused during anchoring. Since `runMain` is not static, an instance of the declaring class is implicitly provided, which is contained in the method header as well. Note finally, that the method header is not a node in the AST but information provided by the method declaration node that the method invocation expression node refers to.

7.2.2 Anchoring algorithm

If the preconditions from section 6.2.2 hold, the following anchoring algorithm will transform the AST – not the source code – into a state that satisfies the postconditions given in section 6.2.3 without violating the invariants specified in section 6.2.4.

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[5,6,7]. Line numbers are not updated.

[5]This step is required to satisfy general invariant 1 stated on page 51 but it does not yet account for all possible complications two of which have been uncovered already. (1) Within a constructor body, the implicit or explicit constructor invocation must be the first statement [GJSB05, 242f.]. I assume that it is possible to insert synthetic local variable declarations to store the value of arguments to the constructor invocations. This would need to happen after step 1 and before steps 2 and 3 in [GJSB05, 322] which is before the fields of the instance have been initialized. The Java Virtual Machine Specification limits operations performed on `this` before the invocation of another initialization method within an instance initialization method to assignments to fields declared for the class of `this` (see [LY99, 147]). Inserting local variable declarations and assignments to them seems hence to be valid at this location. (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.

[6] Note: the ForInit and ForUpdate parts of a `for` statement may contain an anchor. If that is the case, the entire for statement is the one that is rewritten instead of S.

[7]+ tests.Kind1.test80AvoidCollisionWithExistingName()
+ tests.Kind1.test81NoInterferenceWithExistingVariable()
- tests.Kind1.test82NoVarAccessCanReferToGeneratedVariable()

2. Replace[8] the method invocation or class instance creation expression[9] 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*.

3. Rewrite the indirect anaphor *IA* into an access to the local variable *V*.

7.3 Kind 2: Anchoring Based on Fields and Accessors

This section describes an implementation of what has been specified in section 6.3. An anchoring algorithm is given that, if the preconditions from section 6.3.3 hold, will transform the AST – not the source code – into a state that satisfies the postconditions given in section 6.3.4 without violating the invariants specified in section 6.3.5.

Note that the anchoring algorithm for this kind of indirect anaphor is incomplete and to the most part unimplemented.

1. The value of the anchor is obtained in one of the following ways.
   a) If the anchor is a parameter declaration or a local variable declaration, an access *A* to the parameter or local variable will be used to obtain the value of the anchor at run-time. Evaluation of *A* will at run-time happen immediately before the indirect anaphor that is anchored in the declaration is evaluated[10].
   b) If the anchor is an access to a parameter or an access to a local variable, an access *A* created from the declaration of the parameter or local variable will be used to obtain the value of the anchor at run-time.
   c) More ways are to be defined.

2. If the indirect anaphor denotes a field (see preconditions 3a and 4a), it is replaced by a field access expression that is qualified by access *A*.

3. If the indirect anaphor denotes a getter *G* (see precondition 3b), it is replaced by a method invocation expression *M* that does at run-time invoke *G* and *M* is qualified by access *A*.

4. 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 *R* and the following steps are taken to compile the indirect anaphor.

[8]JastAddJ allows a node in the AST to be rewritten into a node of another type or a tree that can contain the original node.

[9]Note that method invocation expressions (see [GJSB05, 440]) as well as class instance creation expressions (see [GJSB05, 423]) have forms that contain a primary expression on the left that can be arbitrarily complex.

[10]The value denoted by *A* will be the same at the time when the execution of the declaration finished and the time when the indirect anaphor is executed. If that was not the case, there would be an assignment to the parameter or local variable between anchor and indirect anaphor that would itself be a suitable anchor and would thus lead to referential ambiguity and a compiler error. Concurrent access to the value of a parameter or the value of a local variable from another thread is equally impossible.

a) A new private final method *M* is declared that has a unique name and is marked as synthetic. *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[11].

b) *AE* is replaced by a method invocation expression that does at run-time invoke *M* and is qualified by access *A*, providing the result of *R* as an argument to *M*.

7.4 Test Case Nomenclature

The implementation was developed test-first with JUnit test cases 76 of which have been developed that are organized using the nomenclature given in table 7.1.

[11] This is necessary because indirect anaphors are expressions i.e. are, in contrast to setter methods, evaluated to a value, not nothing (void). This is especially relevant when the indirect anaphor appears as the left-hand side operand of an assignment expression because it will in the next step replace the assignment expression. Then the value that the indirect anaphor is evaluated to will function as the value that the assignment expression would have been evaluated to.

File	Grouped tests	Focus of the tests in the group
General	01 - 07	General tests on what expressions can be a suitable anchor
	10 - 19b	Types used as part of indirect anaphors
	20	Refential ambiguity due to multiple occurences of a potential anchor (see point 1f on page 47 and section 5.4.3)
	30 - 32	The 2-statements limit (see point 1i on page 47)
	40 - 46	Assignment compatibility of the result type of the anchor to the result type of the indirect anaphor (see section 5.4)
	50 - 53	Multiple anaphors per anchor (see section 5.4.1)
	60 - 64	Syntax of indirect anaphors (see section 5.2)
Kind1	01 - 05	Expressions involving method access of various complexity referred to by indirect anaphors of kind 1
	20 - 22	Expressions involving class instance creation of various complexity referred to by indirect anaphors of kind 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
Kind2	01 - 02	Retrieving and setting the value of fields
	10 - 11	Referential ambiguity due to concurrent suitability of anchors for kind 1

Table 7.1: Grouping of the test cases supplied with the implementation

8 Evaluation and Outlook

At the end of the way from cognitive linguistics to an initial Java-based implementation, before proceeding to do what is yet to be done, time is available to look behind at the value of the current findings. The evaluation in this chapter cannot be a final one, because only the simplest kinds of indirect anaphora have yet been implemented. Hence, this chapter will only contain outlines of a proper evaluation and discussions where evidence is desired. I will evaluate not only what has been implemented but try to analyze potential developments of the general idea of statically-resolved indirect anaphora in naturalistic programming. Therefore, the evaluation starts from the aims laid out in the introduction and will then shift towards applicability of indirect anaphors, the correctness of their resolution and finally discuss compatibility between Jaaa and Java.

Future Work 8.1 (Potential for evaluation) *Areas that shall be included in a full evaluation of statically-resolved indirect anaphora include the following. (1) Fragility: How easily indirect anaphora break upon source code modifications in locations between an indirect anaphor and its anchor and what can be done about this is important to consider. (2) Debuggability: Invisible generated code is bad during debugging because there is no representation of the executed statements in the source that the programmer sees. A dynamic source representation or an intermediate source format after pre-compilation may help. The solution that AspectJ found to this problem is to be considered. (3) Economical advancement: When all or a relevant number of non-trivial kinds of indirect anaphora have been implemented, it will become possible to measure to what extent use of indirect anaphora can shorten the source code of programs. (4) Impact on tools: Tools for refactoring and other tasks will need to be adapted so they can detect and process indirect anaphora. (5) Complexity: How does the compiler implementation scale? (6) Fitness of the transfer: As indicated in section 5.5, the hypotheses underlying the approach of a metaphorical transfer of indirect anaphors from natural language to programming languages are that (a) the cognitive model used correctly describes how humans use natural language, (b) it can be transferred to programming languages and (c) the transfer makes programming languages intuitive as per section 3.2. All three hypotheses shall be proven. (a) is supported by the samples in [Sch00], (b) will be proven by a successful implementation even though the success of an implementation needs to be qualified, (c) requires experiments involving human programmers.*

8.1 Aims

As part of the introduction, section 1.2 lists goals that I wanted to reach as part of my work that is documented here. I will not criticize myself for not getting my text done in time because I did not attempt to reach an outcome well-known in advance. Instead, I will consider to what extent the goals from section 1.2 have been reached.

The first and foremost goal was to work out an access to the topic of natural language programming. In that sense, the discovery of a useful model in cognitive linguistics literature was a success that I expect to bear fruit for future work on the topic. The notion of *programming languages* being a metaphor for *coding systems* lead to the idea of considering indirect anaphora as a metaphor added to programming languages. The metaphor has been developed based on Schwarz-Friesel's cognitive model and provides a good framework to design the current specification and implementation as well as future incarnations of these. The metaphor is expected to be revised as work progresses, in the same way that it has already been revised considerably.

Verification of the proposals through practical applications is yet outstanding, partly because the bottom-up approach did not yet unfold to full extent: the effect that control structures have on indirect anaphora has not yet been considered. The impact of indirect anaphora on inheritance and re-use has, during development of the concept, been minimized, which is positive, though. Continued orientation towards the syntax of Java may help avoid proactive inference (see page 1). From the work done so far it seems that compile-time resolution of indirect anaphora is viable. Specification and implementation are not yet complex enough to hit potential limits, though. It is also not yet possible to compare whether dynamic (i.e. run-time) resolution of indirect anaphora would lead to more possible referents or values denoted by an indirect anaphor.

8.2 Applicability

Will it be possible to make use of indirect anaphora in a relevant number of cases justifying the effort to develop and use indirect anaphora? This question is suited to quantitative analysis if it has been defined (a) what such a case is and (b) how such a case becomes relevant.

Use of indirect anaphora (mentioned as *a case* in (a)) is subject to conditions defined in sections 5.2 and 5.4 and in chapter 6 that have to be met for an indirect anaphor to be successfully anchored and counted as a use of indirect anaphora. Since these conditions were and continue to be in a state of flux, quantitative data that had been collected was discarded and will not be collected before all relevant kinds of indirect anaphora have been implemented and thematic progression is used instead of the 2-statements limit which requires the specification of the breadth of themes (see future work item 2.10 on page 24). Additionally, potential interferences between multiple indirect anaphors need to be considered, especially when an indirect anaphor can refer to other indirect anaphors. The impact of control structures and complex statements like *for* statements may also be relevant to the development of indirect anaphora. Implementation of direct and complex anaphora may defer quantitative analysis further[1] or invalidate the results at some point because new kinds of anaphora have (partly) the same syntax as all other kinds but will affect when anaphora are applicable or when different kinds of anaphora apply. Having mentioned the syntax of anaphors, it shall be noted that the syntax of anaphors may change with time. Implementations of the Pegasus project include attribute specifications as part of anaphors

[1]When only indirect anaphora have been implemented, only those local variables can be replaced by anaphors that are accessed exactly once after having been initialized. This is due to the fact that an indirect anaphor never refers to the node of an anchor (this would be co-reference as in the case of the direct anaphor shown in figure 2.1 on page 10) but to another node related to the node of the anchor. I.e. an indirect anaphor can be used to gain access to a node, but cannot repeatedly access that node. This is why direct anaphors complement indirect anaphors, because the direct ones can (repeatedly) take up what has been made accessible to them by the indirect ones.

and this will be necessary in Jaaa as well. Consider an object-oriented implementation of a tree in which all nodes are instances of the same class that provides accessor methods to its direct children and as well as to its indirect children. These methods will return collections of the same type, leading to a compiler error when trying to refer anaphorically to (the return value of) one of the methods using the collection type. A form of anaphor may be introduced that allows an attribute or name to be combined with the type used in the indirect anaphor, which is a change in the syntax of anaphors.

When uses of indirect anaphora can be counted, not all of them will be relevant. A relevant use of indirect anaphora is one that makes it worth to use an indirect anaphor instead of existing features of Java. E.g. to replace local variables by indirect anaphors makes sense only if one or more of the following criteria apply (1) the source code becomes shorter, and/or (2) the source code becomes easier to understand by programmers. While (1) can easily by measured by counting the number of non-whitespace characters in source code, it is more difficult to judge how hard it is for programmers to understand the meaning of source code. Discussion of the examples in sections 6.2.1 and 6.3.2 pointed out that experiments are needed for this. The reader-centric model of anaphora resolution from chapter 2 implies that this judgement is relative to individual programmers what would need to be accounted for in the experiment design. There may be multiple experiments investigating different tasks as part of which programmers try to understand code because with different tasks the code may be understood from different perspectives: e.g. (1) reading the source code in order to comprehend what it is supposed to do or (2) reading the source code augmented with run-time values during debugging to find out why the program behaves differently than expected after reading the source code without run-time values. Judgement of the intelligibility of code with indirect anaphors is not only relative to the programmer and task, but also dynamic. The interaction of indirect anaphora and learning the structure of an application will require attention. In Java source code the structure of an application is either hard to miss or dissolves in detail. Indirect anaphora may remove the dissolution in detail but may as well make it harder to get the structure because the underspecification of what is assumed to be known is how indirect anaphora shortens texts. A help out of this problem may lie in dynamic underspecification: the original author may underspecify, but new readers could be presented an elaborated version of the source code that may become underspecified when it is safe to assume that they learnt the structure of the application.

8.3 Correctness of Resolution

Not only the relevance of a use of indirect anaphora is dependent on the judgement of programmers. The correctness of the resolution of an indirect anaphor does depend on the individual programmer as well. While a compiler will always be able to obey to any semantics considered normative as long as it is correctly implemented, a human programmer is supposed to benefit from naturalistic programming because the new means of reference in Jaaa are supposed to work like means of reference that the programmer does already know from natural language. This can, however, not be ensured by implementing a specification of semantics. Experimental evidence is necessary showing that a relevant number of programmers confirm that the referents found for indirect anaphors are the ones that they expect (see 3.2) and that proactive inference does not lead programmers to expect referents that the compiler does not refer to (see page 1).

8.4 Compatibility

Since Jaaa is an artifact of the evolution of programming languages, the question of compatibility arises. The compatibility of different versions of Jaaa will not be considered because Jaaa is not intended to be used except for furthering its development. Compatibility between Java and Jaaa is important, though. There are two key questions concerning compatibility between Java and Jaaa.

Can Java code be re-used in Jaaa programs? Indirect anaphors are syntactically distinguished by the leading "." that does not occur in a valid Java program i.e. the semantics of Java are not modified in Jaaa. Hence, a Jaaa compiler must be able to compile Java programs because Jaaa is Java plus indirect anaphors which can be identified during parsing already. Due to that, Jaaa programs can re-use Java source code as well as Java class files.

Can Jaaa code be re-used in Java programs? Since Jaaa adds another syntactic element to Java, a Java compiler will not be able to compile Jaaa source code but produce a syntax error instead. Java programs can, however, re-use Java class files produced by a Jaaa compiler because they adhere to the Java class file format. The description of APIs using the Javadoc tool should also be possible for Jaaa source code, since indirect anaphors do not surface at (class) interfaces that are relevant for inheritance and composition. This aspect may be complicated if scripts are implemented in Jaaa which will not be associated to a class in the traditional sense anymore (see future work item 5.1 on page 41).

8.5 Summary

After listing some potential for further evaluation, it was found that the focus on cognitive linguistics that serves as the basis of this work will continue to bear fruit. Further judgements are not yet possible, since the implementation is yet too early to be influenced by control structures considerably. Similarly, it is not yet possible to compare statically-resolved anaphora implemented here to the run-time resolution implemented as part of the Pegasus project. Whether or not indirect anaphors are well applicable to programming problems is not yet to be judged, because the syntax and semantics of indirect anaphors are expected to continue to change and be impacted by introduction of direct and complex anaphors. A judgement of the applicability of indirect anaphors is also not trivial, since it is relative to individual programmers and tasks and needs to incorporate the dynamics of learning. Related to applicability is correctness of resolution: whether the referents that the compiler resolves are the ones that programmers expect. It was finally highlighted that while Java sources and binaries can be re-used in Jaaa, only Jaaa-binaries can be re-used in Java.

9 Summary and Conclusions

The idea of static resolution of indirect anaphors draws upon a number of existing works. The earliest ones reach back to 1965: that year Jean E. Sammet, proposed the bottom-up approach that I took on and there was even the idea of adding a pronoun to ALGOL. Lopes et al. renewed interest in adding anaphora to programming languages and the Pegasus project finally implemented them, resolving them at run-time. Hence, I chose to add a more complicated form to Java (indirect anaphors) and to perform anaphora resolution at compile-time.

Definitions of reference, proper and common 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 include a modular theory of mental representations, three-tier semantics that involve conceptual and semantic knowledge, a text-world model and how readers elaborate this model. Focus and activity were described, that serve to explain the dynamic limitation of searches for antecedents. Schwarz-Friesel's process-oriented cognitive model of anaphora resolution was exemplified, which distinguishes four kinds of indirect anaphora based on the knowledge involved in their resolution: anchoring based on thematic roles, meronymy-based, schema-based and inference-based anchoring. An corresponding algorithm for selecting a single suitable anchor from a set of potential anchors that involves the economical selection of the kind of anchoring to be applied was reproduced as well. 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, forms of knowledge representation, the (ad-hoc) creation of concepts and conceptual schemata as well as inference processes and their relation to logical inference require further reading, as do issues like the depth of conceptual decomposition, representation and granularity of theme and rheme as well as thematic progression that are, to my knowledge, open questions in theoretical 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.

A metaphor of indirect anaphora was developed through a transfer from natural language to a dialect of Java called Jaaa and was elaborated at the levels of syntax, semantics and pragmatics. The metaphor incorporates cognitive foundations and Schwarz-Friesel's model of indirect anaphors and their anchoring. It was argued that textual and memorized knowledge is identical in Jaaa and that an abstract syntax tree of Jaaa source code functions as a text-world model. I did not yet understand defaults well enough to fully integrate them into the metaphor. Similarly, thematic progression is not yet implemented. However, it was found that referential ambiguity can in some cases be automatically reduced in Jaaa. That the metaphor of indirect anaphora in programming languages is rooted in cognitive linguistics suggests that it makes programming languages more natural (i.e. known or easy to learn) which is a hypothesis that is yet to be proven.

Three kinds of indirect anaphors have been proposed for Jaaa, giving pre- and postconditions and invariants. The first one, which has already been implemented, was derived from anchoring based on thematic roles and accesses the value returned or created by a method invocation expression or class instance creation expression. The second kind has been implemented in minor parts only. It is derived from meronymy-based and schema-based anchoring at the same time because Java does not allow for a distinction between parts of an instance and objects directly related to it. Indirect anaphors of the second kind can appear as the left-hand side operand of assignment expressions which makes them function as variable assignment or setter invocation. In all other positions, they will function as variable access or getter invocation. An annotation may be introduced that marks unconventional accessors. A third kind of indirect anaphors is to be modeled after inference-based anchoring.

The prototypical implementation of the proposed indirect anaphors is based on JastAddJ, a modular, aspect-oriented Java compiler whose implementation uses attributes and rewrites. The implementation is quite limited: it has not yet been tested with real-world source code but with artifical code samples only. Nothing not yet covered by the test cases can be expected to work. The implementation of the first kind was introduced with a discussion of the parsed and the transformed abstract syntax tree and has an anchoring algorithm that inserts, assigns and accesses a local variable. An incomplete anchoring algorithm is given for the second kind that is to the most part unimplemented. An overview of the test cases is provided.

The evaluation of this work is more of an outlook. While the aim of getting an idea of the field was achieved and the focus on cognitive linguistics proved useful and is expected to bear fruit, implementation is still so early that the impact of control structures cannot yet be judged and a comparison to run-time resolution is yet unfeasible. Similarly, the applicability of indirect anaphors to programming problems depends on upcoming changes to the syntax and semantics of indirect anaphors and on the introduction of direct and complex anaphors. A judgement of the applicability of indirect anaphors is also not trivial, since it is relative to individual programmers and tasks and needs to incorporate the dynamics of learning. Related to applicability is correctness of resolution that expresses whether the referents that the compiler resolves are the ones that programmers expect. It was finally highlighted that while Java sources and binaries can be re-used in Jaaa, only binaries but not sources of Jaaa can be re-used in Java. The next steps will hence be to complete the implementation of indirect anaphors of kind 2. Thereafter indirect anaphors of kind 3, direct anaphors and thematic progression (no order implied) will likely be researched, specified and implemented.

Bibliography

[Bus98] Carsten Busch. *Metaphern in der Informatik*. Deutscher Universitäts-Verlag, 1998.

[Cho57] Noam Chomsky. *Syntactic Structures*. Mouton & Co., The Hague, Paris, 1957.

[CKSF07] Manfred Consten, Mareile Knees, and Monika Schwarz-Friesel. The function of complex anaphors in texts. 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 81–102. John Benjamins, Amsterdam, 2007.

[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.

[EH06] Torbjörn Ekman and Görel Hedin. The jastadd system – modular extensible compiler construction. In Torbjörn Ekman, editor, *Extensible Compiler Construction (Diss.)*, pages 61–73. 2006.

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 884–885, New York, NY, USA, 2007. ACM.

[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. *The Java(TM) Language Specification*. Addison Wesley, 3rd 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 Ruaqiya Hasan. *Cohesion in English*. Longman, Harlow, 1976.

[Hig67] Bryan Higman. *A comparative study of programming languages*. Macdonald, London, 1967.

[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.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[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$^+$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.