

Representing Ontological Structures in CLOS, Java and FAST

Bachelor Thesis in Cognitive Science

by Fabian M. Suchanek

University of Osnabrück / Germany

First Supervisor: Dr. habil. Helmar Gust
Second Supervisor: Dr. habil. Ute Schmid

30 September 2003

Abstract

The issue of knowledge representation is one of the core areas of artificial intelligence. One of the most common forms of world knowledge representation is an ontological model, in which concepts are organized in a hierarchical structure. Sometimes, such an ontological structure is to be used in a software system. The question arises whether it is possible to represent the structure directly by the programming language. Since object-oriented programming languages base on similar paradigms as the ontological model, they seem predestined for this purpose. The goal of this paper is to compare the object-oriented programming languages CLOS, Java and FAST with respect to their capability of representing ontological structures. To do so, a detailed introduction into the computational interpretation of the ontological model will be given. Then, different ontological problems of increasing complexity will be presented. For each of them, it will be illustrated and assessed in how far the programming languages are capable of solving them. Special attention will be paid to the problems of multiple inheritance. It will be investigated in how far CLOS, Java and FAST are appropriate means for the representation of ontological structures.

Zusammenfassung

Die Repräsentation von Wissen ist eines der Hauptgebiete der künstlichen Intelligenz. Eine häufig gebrauchte Form der Wissensrepräsentation ist ein ontologisches Modell, in dem Konzepte in Hierarchien angeordnet sind. Manchmal soll solch eine ontologische Struktur in einem Software-System benutzt werden. Dann stellt sich die Frage, ob es möglich ist, die Struktur direkt in der Programmiersprache darzustellen. Da Objekt-orientierte Programmiersprachen auf ähnlichen Paradigmen basieren wie ontologische Modelle, scheinen sie für diese Aufgabe prädestiniert. Das Ziel dieser Arbeit ist der Vergleich der Objekt-orientierten Sprachen CLOS, Java und FAST in Hinblick auf ihre Fähigkeit, ontologische Strukturen zu repräsentieren. Dazu wird zunächst eine detaillierte Einführung in das ontologische Modell der Informatik gegeben. Sodann werden verschiedene ontologische Probleme mit wachsender Komplexität vorgestellt. Für jedes von ihnen wird detailliert untersucht, inwieweit die Sprachen Möglichkeiten zu einer Lösung bieten. Dabei wird auch auf die Probleme der Mehrfachvererbung eingegangen. Es wird herausgefunden, inwiefern CLOS, Java und FAST für die Darstellung ontologischer Strukturen geeignet sind.

Acknowledgements

This is my first scientific paper. I could not have written it without the support of my supervisor Dr. habil. Helmar Gust. I would like to thank him for the helpful discussions and his fruitful criticisms. Furthermore, Saskia Nagel deserves my thanks for critically reading this thesis.

Table of Contents

1. The Philosophical and Computational Background	4
1.1. Introduction	4
1.1.1. Motivation	4
1.1.2. Relation to Other Disciplines	4
1.1.3. Outline of the Parts and Chapters	5
1.2. The Computational Model of Ontology	6
1.2.1. The Assumption of Discrete Entities	6
1.2.2. The Assumption of Attributes	7
1.2.3. The Assumption of Concepts	8
1.2.4. Objects, Classes and Fields	10
1.3. The Programming Languages CLOS, Java and FAST	11
1.3.1. CLOS	11
1.3.2. Java	12
1.3.3. FAST	13
1.3.4. Classes in CLOS, Java and FAST	13
1.3.5. Criteria for the Comparison of the Languages	14
2. Basic Concepts of Object-Orientation	15
2.1. Classes and Attributes	15
2.1.1. Classes	15
2.1.2. Shared Attributes	16
2.1.3. Private Attributes	17
2.1.4. Chapter Summary	17
2.2. Methods	19
2.2.1. Normal Methods	19
2.2.2. Multi-Methods	19
2.2.3. Variable Instance Methods	21
2.2.4. Accessor Methods	22
2.2.5. Chapter Summary	23
2.3. Object Creation and Attribute Access	24
2.3.1. Object Creation	24
2.3.2. Object Initialization	24
2.3.3. Access to Attributes	25
2.3.4. Access to Shared Attributes	26
2.3.5. Method Calls	27
2.3.6. Chapter Summary	27
3. Inheritance	28
3.1. Single Inheritance	28
3.1.1. Sub-Classes	28
3.1.2. Type Refinement	29
3.1.3. Overwriting Attributes	30
3.1.4. Overwriting Shared Attributes	31
3.1.5. Overwriting Methods	31
3.1.6. Inference Mechanisms	32
3.1.7. Chapter Summary	33
3.2. Multiple Inheritance	34
3.2.1. A Sister Concept in CLOS and FAST	34
3.2.2. A Common Child Concept in CLOS and FAST	34
3.2.3. Interfaces in Java	35
3.2.4. A Sister Concept in Java	36
3.2.5. A Common Child Concept in Java	36
3.2.6. Chapter Summary	37

3.3. Problems of Multiple Inheritance	38
3.3.1. Ambiguous Inherited Attributes	38
3.3.2. Multiply Inherited Attributes	39
3.3.3. Inference Mechanisms with Multiple Inheritance	40
3.3.4. Chapter Summary	42
4. Final Reflections	44
4.1. Conclusion	44
4.2. Outlook	45
Appendix	46
A. Program Code	46
A.1. Program Code in CLOS	46
A.2. Program Code in Java	48
A.3. Program Code in FAST	51
B. Bases of Relations	54
C. References	55

1. The Philosophical and Computational Background

1.1. Introduction

1.1.1. Motivation

Many tasks in the area of information technology require the representation of world knowledge. Robots need knowledge of their environment, translation systems need knowledge of word meaning relations, and intelligent database systems need knowledge of their domain. All these applications require the description of a real world domain by formal means. Many models for knowledge representation have been proposed and used.

In the last decades, models involving concept hierarchies became increasingly important (cf. Kopp, 1996). In 1968, Ross Quillian first proposed *semantic networks* for the representation of word meaning (1968). Semantic networks are graphs, the nodes of which are nouns, adjectives and verbs. The arcs of the graph reflect relations between the word meanings. These graphs can already display a hierarchy of words, in which general terms are located higher than less general terms. 7 years later, Marvin Minsky proposed so-called *frame-systems* for the representation of situations (Minsky, 1975). A frame is a unit, which stands for a situation or an object. It possesses slots, which store information related to the object. Similar to the nodes of semantic networks, frames can be arranged hierarchically. In 1985, Ronald Brachmann and James G. Schmolze developed *KL-ONE* (1985). KL-ONE is a formal calculus for concept representation, which can be seen as a logical implementation of the frame model (Weiermann, 2000). All of these theories contribute to a basic understanding of concepts, which is widely accepted today. I will call this vague model the *ontological model*. In the ontological model, the world is seen as a set of objects. These objects, called *entities*, have certain *properties*. According to these properties, similar entities are classified into *categories* or *concepts*. These concepts are arranged in hierarchies, in which more general concepts dominate less general concepts. These hierarchies will be called *ontological structures*. The ontological model has proven to be a reliable framework for many variants of world knowledge representation. It is for instance used in one of the world's largest knowledge bases, Cyc (Reed and Lenat, 2002). The ontological model will be presented in detail in this paper.

Often, one is not satisfied with modeling and representing the knowledge, but one would like to apply the representations in algorithms. Robots use their knowledge to perform certain actions, translation systems use their knowledge to read and create documents, and database systems use their knowledge to display data and answer queries. As the representations are to be used by computer programs, it seems promising to employ the programming languages themselves for the representation and modeling of world knowledge. If one focuses on the ontological model, the question is whether a programming language can represent an ontological structure. Since object-oriented languages share many aspects of the ontological model, they appear most appropriate for the task. This paper focuses on three of them: The first one is CLOS, the object-oriented extension of the most common language of artificial intelligence LISP. The second is Java, which is about to evolve to a new standard programming language in computer science. The third is FAST, which is a student's project by me. These languages will be compared regarding their capability of ontological knowledge representation.

1.1.2. Relation to Other Disciplines

Although this paper focuses on the application of the ontological model in artificial intelligence, the idea of ontological structures appears in many other disciplines. It first emerged in Philosophy. The philosophical sub-discipline concerned with the nature and relations of being is called *Ontology*¹. The desire to classify the objects of the world can be traced back to Aristotle, who also coined the term *category* (Sowa, 2000). The first concept hierarchy is attributed to the philosopher Porphyry, who lived in the third century AD. Later, concept hierarchies were used by many other philosophers, among them Immanuel Kant, Franz Brentano, Charles Sanders Peirce and Alfred North Whitehead. Today, John F. Sowa is one of the most important researchers on ontological structures.

¹ I follow Guarino's distinction of "Ontology" (meaning the discipline) and "ontology" (meaning a certain categorization) (1998).

In psychology, the ontological model serves as a model for human information processing. Allan Collins and Ross Quillian noted that semantic networks satisfy the principle of *cognitive economy*, because they provide a very efficient way of storing information (cf. Eysenck and Keane, 2000). By experiments, they could show that semantic networks could predict the reasoning time of humans. Hence, it seemed possible that human rational thinking can be interpreted as a semantic network. Later, Eleanor Rosch (1978) took up the idea of ontological structures. She established three levels of generalization in the ontological hierarchy. Her experiments proved that these levels are distinguishable in human category thinking. These studies serve as a support for the psychological relevance of the ontological model.

In linguistics, ontological structures are used to capture the meaning of words. The set of entities, to which a word applies, is called the *extension* of this word. Basing on this notion, Lyons arranges words in a hierarchy, so that the extension of a word always includes the extensions of its sub-ordinate words (1977). Thanks to these structures, Lyons is able to define the semantic relations of *hyponymy* and *synonymy*. Thereby, the ontological model contributes to the study of semantics.

The most prominent domain for the application of the ontological model is Biology. In the 18th century, the biologist Carolus Linnaeus categorized all known living beings in his "Systemae Naturae". This work formed the basis of the modern classification of animals: Each individual animal belongs to a *species*. According to their properties, similar species form a *genus*. Similar genus form *families* and this generalization continues up to *orders*, *classes* and *kingdoms* (Minelli, 1993): The classification system used today in Biology is a huge ontological structure.

The ontological model also had an influence on one of the youngest sciences, computer science. Programming languages were originally designed for simple instruction. It became soon apparent that structuring tools were needed for programs that were more complex. Consequently, concepts such as functions, abstract data types and modules evolved. In the late sixties, it was discovered that ontological hierarchies might also serve as structuring tools. In 1967, Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Centre designed Simula, the first language to support objects and classes (Holmevik, 1995). Thereby, the paradigm of *object-orientation* was introduced in computer science. In fact, object-orientation proved to be an extremely useful means for structuring programs, as it supports data abstraction, encapsulation and the factorization of information (Ducourneau and Habib, 1991). The application of object-orientation will be the topic of this paper.

1.1.3. Outline of the Parts and Chapters

This thesis is divided into four parts. The first part covers the philosophical and computational background for the language comparison. Following this introduction, chapter 1.2. will introduce the *computational model of Ontology*, which is the ontological model used in computer science. Then, chapter 1.3. will present the programming languages CLOS, Java and FAST. An overview of the history, paradigms and applications of these programming languages will be given. Furthermore, chapter 1.3. will fix the criteria for the comparison of the languages. The second part of this thesis is devoted to the main investigation: Different ontological problems will be posed. It will be shown how solutions could be implemented in CLOS, Java and FAST. It will be compared and evaluated in detail in how far the languages provide the means for a convenient solution. The concept of *tigers* will serve as a running example throughout the paper. At first, in chapter 2.1., this simple concept will be implemented along with some attributes. Then, actions will be associated to the concept in chapter 2.2. In chapter 2.3., it will be shown how instances of the concept can be created. The third part of this paper illustrates the phenomenon of inheritance: In chapter 3.1., a sub-concept of *tiger* will be created, namely the concept of *pet tigers*. The inference problem of Ontology will be presented and solved in CLOS, Java and FAST for this simple case. Chapter 3.2. will concentrate on multiple inheritance. It will be shown to what respect the programming languages support concepts with more than one super-concept. In chapter 3.3., complex problems resulting from multiple inheritance will be presented. It will be investigated how the programming languages cope with them. The fourth and last part of this thesis contains the conclusion and the outlook. It will summarize the results of this paper and conclude about the usefulness of CLOS, Java and FAST for ontological knowledge representation.

1.2. The Computational Model of Ontology

In this chapter, the ontological model used in computer science will be introduced. I will call this model the *computational model of Ontology*. It shall form both the basis and the frame of this paper. We will start from scratch and think about the basic building blocks of the world. As Willard Van Orman Quine put it, the study of Ontology tries to answer the simple question of "What is there?" (qtd. in Sowa, 2000). The answer, however, is of immense complexity. Two a priori problems arise immediately: First, any thought on existence exists itself. Thus, Ontology faces the inherent problem of *self-reflectivity*; it is the subject of its own research (Bibel, Hölldobler and Schaub, 1993). Second, Ontology has to define the basic notions, which any theory (including itself) bases on. Thus, it cannot fall back upon terms, which have already been defined in order to define new terms. Strictly speaking, it cannot define any term without creating cycles. I will pass over this problem by defining the basic terms in a non-formal language, presupposing a certain common sense understanding of English. Many other assumptions will have to be made in order to establish the computational model of Ontology.

1.2.1. The Assumption of Discrete Entities

The notion of *entity* is the most general one at all: Anything is an entity, whether it is concrete or abstract, an object or a structure, a state or an action, a thought or a person. Mostly, the notion of *entity* is closely linked to the notion of *existence* (Merriam-Webster, 1994; Sowa, 2000). This entails the need to know whether something has being in reality in order to call it *entity*. There are things, though, the existence of which is uncertain (such as souls), false (such as dragons), improvable (such as God), hypothetic (such as phlogiston) or dependent on the reading of existence (such as fictional characters). I will therefore understand the term *entity* independent of *existence* and call everything an entity, whether it has being or not.

The term presupposes the isolation of discernable elements. By contrast, the world is made up of a variety of continuous flows, graduations and transitions (Bateman, 1993). Blobs of rain, for instance, fall down, join in a puddle and may be splattered by a passing car to form new drops (Sowa, 2000). The discernment of distinct entities is at least subject to discussion, if not impossible. The terminology of discrete entities is thus just an approximation of reality. Through this grid, one sees only distinct beings.

The convention of distinct entities leads to another difficulty, which is the *problem of parts and wholes* (Varzi, 1998). If a thing (like a window) consists of two things (a frame and a pane), then this could be two entities (a frame and a pane), one entity (a window) or three entities (a pane, a frame and a window). Although all alternatives might seem plausible, there will be three entities in the computational model, namely the aggregate and its two components. However, the model cannot cope that easily with the puzzle of the Greek philosopher Eubilides (Varzi, 1998): If one takes away one molecule of an object, will there still be the same object? If it is still the same object, this invites one to take away more molecules until the object disappears. If it is another object, this forces one to accept that two distinct objects occupy the same spatio-temporal location: The whole and the whole without the molecule. In a computational ontology, actions of this kind are prohibited. A computational model is a snapshot of the world, in which entities exist and may only be created and destroyed as wholes.

Another question is concerned with identity: In the course of time, parts of an entity may be replaced by other parts. The question is whether the whole is still the same entity or not when all parts have been replaced (Varzi, 1998). The ancient philosopher Theseus uses the example of a ship: Its old planks are constantly being substituted. One day, the whole ship has been replaced and Theseus asks, "Is it still the same ship?" It seems as if the structure itself constitutes the entity rather than its components. In a computational ontology, identifiers are used which refer to entities. This technique fixes identity from outside and thus the problem cannot appear.

A computational ontology possesses a set \mathfrak{S} of *entity identifiers*. Again, the notion of a set is not without its problems. Set theory forces the distinction between a singleton set and an element, it possesses the ontological extravagance of being able to create something out of nothing (namely the

empty set) and it is not free of paradoxes². Furthermore, if this set contains identifiers for all entities, it has to contain one for itself as well. Even worse, it has to contain an identifier for each identifier, since also names are entities. In order to deal with all these problems, I confine myself to a finite set of observed entities, the domain. Now, the set contains a finite number of identifiers, which refer to the entities of the domain. To simplify, I will talk of *entities* although their identifiers are meant.

1.2.2. The Assumption of Attributes

Entities have certain qualities: A leaf is green, a dog is an animal and person X may be fat. The assignment of these qualities is seldom indisputable. Not only person X could be of another opinion concerning his or her fatness, but also different people could be of different opinions concerning the greenness of a leaf. Leaves may be more or less green³ and it is not clear whether the green of leaves is the same as that of a traffic light, for instance. These are issues of semantics and pragmatics, which a computational model does not deal with. Furthermore, to avoid the problems of subjectivity and epistemology, a computational model assumes an omniscient perspective. In this view, all qualities are known and well defined⁴.

I will now examine the ontological status of properties. As the definition of *entity* embraces all beings, the ideas of *greenness*, *animalness* and *fatness* are certainly also entities. It is not clear, though, how they are related to the entities to which they apply. One might imagine unary predicates in the sense of First Order Logic or one might define functions, which map an entity onto a property. All of these approaches are artificial and arbitrary choices to describe the phenomenon of properties. They explain neither why an entity has a property nor what it means to have a property nor what the nature of *property* is.

In the computational model, another arbitrary way is chosen: All properties are described by a set \mathfrak{R} of relations on $\mathfrak{E} \times \mathfrak{E}$. These relations are called *attributes*. They usually bear the name of qualities. The first argument holds the entity described. It is therefore called the *domain of the attribute*, noted **Dom**. The second argument holds the value of the quality. One writes

```
color(leaf, green)
is-a(dog, animal)
build(person_X, fat)
```

This notation follows the classical idea of attribute-value-pairs. All values are entities and they are part of the domain. Attributes are usually not part of the domain, although they are entities. If an entity is an element of the domain of an attribute, then this entity is said to *have* this attribute.

This approach suffers from a severe shortcoming: The formalization is not unique. Nothing prevents one from interpreting **green** as a quality and **true** as its value, or **being** as the quality and **green** as its value or **color** as the quality and **its_color** as the value, where **its_color** has the property of being a green color. These are matters of design of a computational ontology. As a general guideline, attributes are chosen such that they have only one value for one entity. Furthermore, one tries to minimize the number of attributes and the number of entities.

By definition, attributes are binary relations and may thus express any connection between two entities. Attributes are also able to express unary predicates. Any statement of the form **a(x)** can be transformed to an attribute statement of the form **a(x, true)**. This is a way computational models can simulate properties in the sense of First Order Logic⁵. If an n-ary relation is to be modeled, a new entity has to be introduced, which bears the name of the relation. This new entity is connected by attributes to all n entities concerned.

Relations may be defined by extension or by intension. In the former case, the elements are given by pure enumeration, whereas in the latter case, the elements are given by the "meaning" of the relation. The intensional definition is particularly interesting if the relation is to be used on other domains or in

² See (Smith, 1998) for a detailed criticism.

³ The theory of Fuzzy Logic treats these issues of vagueness.

⁴ Some logicians (among them Jan Łukasiewicz) have proposed three-valued logics, which allow calculating with unknown values.

⁵ Other possibilities would be lists of properties or properties, which store the entities to which they apply.

other possible worlds. Since the starting domain of a computational model is fixed and finite, both approaches lead to the same result here.

1.2.3. The Assumption of Concepts

Up to now, the model consists just of entities and relations. In order to structure the domain, an ordering relation would be useful. Two attributes seem predestined: The **part-of** relation and the **is-a** relation, which are reflexive, transitive and anti-symmetric. It has been reasonably argued, that the **part-of** relation would be the more fundamental one (Varzi, 1998), but for computational ontologies, the **is-a** relation is considered more useful. It constitutes a partial ordering on \mathfrak{I} . In terms of logic, this means:

$$\begin{aligned} & \text{is-a} \subset \mathfrak{I} \times \mathfrak{I} \\ & \forall x: \quad \text{is-a}(x,x)^6 \\ & \forall x,y,z: \text{is-a}(x,y) \wedge \text{is-a}(y,z) \Rightarrow \text{is-a}(x,z) \\ & \forall x,y: \quad \text{is-a}(x,y) \wedge \text{is-a}(y,x) \Rightarrow x = y \end{aligned}$$

Note that these formulas constitute necessary conditions for the **is-a** relation, but do not define it. This paper relies fully on the linguistic meaning of **is-a**. The formulas just serve as a formal support. One might also think about defining **is-a** and the notion of concepts by logical axioms. These axioms could express concept membership, sub-ordination of concepts and attribute propagation. However, it is difficult to ensure that the relations satisfying the axioms are the semantic relations of Ontology, because any artificial relations fulfilling the axioms might be invented. Hence, in order to ensure the semantic adequacy of the computational model, it has been chosen to rely on the linguistic meaning of **is-a** for this paper. Furthermore, the **is-a** relation is here assumed to hold in sentences like "Bonzo is a dog" as well as in sentences like "A dog is an animal". Thus, this relation includes the relations commonly termed **kind-of** and **instance-of**. For this paper, **is-a** has been chosen, because it does not presuppose a definition of concepts. Here, the whole terminology of instances and concepts will be deduced from the general **is-a**.

Each binary relation has a *basis*, i.e. a minimal subset, the transitive closure of which is the original relation. Since the **is-a** relation is a partial order, it is acyclic and its basis is unique (cf. appendix for a proof). I will denote the basis by is-a^- . This relation is an irredundant subset of **is-a**, which only contains those elements, which cannot be constructed by transitivity.

Many definitions for the term *concept* have been proposed. The computational model uses a very simple one: A concept is an entity, which can fill the second place of the **is-a** relation. This definition is to be understood in the linguistic sense of **is-a** and without taking into account reflexivity. Thus, **dog** would for example be a concept, as it is possible to say, "Bonzo is a dog". Analogously, **animal** would be a concept, as one can say, "A dog is an animal". By contrast, **Bonzo** would not be a concept. As the definition bases on the linguistic possibility, **dog** would also be a concept if **Bonzo** did not belong to the domain. Concepts form a subset of the set of entities. If a concept is related by **is-a** to another concept, then the latter is called *more general* than the former. In the example, **animal** would be more general than **dog**. The more general concept is called a *super-concept* of the less general one. A concept is a *direct super-concept*, if the relation cannot be constructed by transitivity. The notions *more special* and *sub-concept* are defined accordingly. Presupposing the predicate **concept**, this can be formalized as

$$\begin{aligned} & \forall x,y: \text{is-a}(x,y) \wedge x \neq y \Rightarrow \text{concept}(y) \\ & \forall x,y: \text{subconcept}(x,y) :\Leftrightarrow \text{concept}(x) \wedge \text{concept}(y) \wedge \text{is-a}(x,y) \\ & \quad \quad \quad \wedge x \neq y \\ & \forall x,y: \text{direct-subconcept}(x,y) :\Leftrightarrow \text{subconcept}(x,y) \wedge \text{is-a}^-(x,y) \\ & \forall x,y: \text{superconcept}(x,y) :\Leftrightarrow \text{subconcept}(y,x) \\ & \forall x,y: \text{direct-superconcept}(x,y) :\Leftrightarrow \text{direct-subconcept}(y,x) \end{aligned}$$

⁶ For convenience, it is implicitly assumed that all lower-case quantified variables are elements of \mathfrak{I} and all upper case variables belong to \mathfrak{R} .

The set of concepts and the **direct-subconcept** relation define a directed acyclic graph. This graph is called the *concept graph*⁷. An element of \mathfrak{S} , which is not a concept, is called an *instance*. An instance must be directly related to exactly one concept⁸. One talks of a *direct instance of a concept*. By transitivity, the instance is also an instance of all super-concepts. This definition prohibits so-called *meta-concepts*. A meta-concept is a concept, the instances of which are concepts. If one demands that every concept be an instance of a meta-concept, then meta-concepts are themselves instances of meta-concepts. Meta-concepts can be used to reason about the properties of concepts themselves. They may for example limit the number of instances of a concept. Furthermore, they can be used to express *collective properties* of the instances. These are properties, which apply to the set of instances, but not to each of them. Since meta-concepts would exceed the frame of this paper, the computational model used here will be restricted to instances and concepts⁹. The result is a simple model, in which the domain is partitioned into instances and concepts:

$$\begin{aligned} \forall x: & \quad \text{instance}(x) :\Leftrightarrow \neg\text{concept}(x) \\ \forall x,y: & \quad \text{instance}(x,y) :\Leftrightarrow \text{instance}(x) \wedge \text{is-a}(x,y) \wedge x \neq y \\ \forall x,y: & \quad \text{direct-instance}(x,y) :\Leftrightarrow \text{instance}(x,y) \wedge \text{is-a}^-(x,y) \\ \forall x: & \quad \text{instance}(x) \Rightarrow \exists!y: \text{direct-instance}(x,y)^{10} \end{aligned}$$

Having selected the **is-a** relation for structuring the domain, I will no longer take **is-a** as an attribute in a strict sense. It may then be demanded that all attributes have no more than one value for one entity¹¹. If an attribute has multiple values, then these values need to be compiled to a collection, which counts as a single entity. Furthermore, it is assumed that all direct instances of a concept share the same attributes. In this case, the concept is said to *have* these attributes. It is postulated that a sub-concept has all attributes of all of its super-concepts; it is said to *inherit* the attributes. Consequently, a super-concept has a (possible empty) subset of those attributes, which are common among all of its direct sub-concepts. Due to this terminology, one can take the viewpoint that a concept is characterized by its attributes¹². The fact of concepts having attributes does not imply the existence of fixed values for these attributes on the level of concepts. Technically speaking, these postulations imply essentially

$$\begin{aligned} \forall R,x: & \quad |\{y \mid (x,y) \in R\}| \leq 1^{13} \\ \forall R,c,x: & \quad \text{direct-instance}(x,c) \wedge x \in \text{Dom}(R) \Rightarrow \{y \mid \text{instance}(y,c)\} \subseteq \text{Dom}(R) \end{aligned}$$

The formalization of concepts is not definite: Due to the flexibility of the **is-a** relation, properties might also be considered concepts. To state for example that Bonzo is brown, Bonzo might be considered an instance of the concept **brown-entity**. This choice is up to the modeler. As a general guideline, concepts should have more attributes than their super-concepts. Assume that **brown-entity** is a sub-concept of **physical-entity**. Then **brown-entity** will not have more attributes than **physical-entity**, since there are no new attributes, which are shared by all sub-concepts of **brown-entity**. As a result, the quality of being brown would rather be considered a property.

Nevertheless, a computational ontology would allow the modeler to consider **brown** a concept. This is due to an *extensional view* of concepts: Computational ontologies model reality; the modeler designs concepts and assigns entities to concepts. This view is to be distinguished from an *intensional view* of concepts, which is often taken in philosophy. This view assumes that concepts are given by their intrinsic attributes and that the true interrelations of entities and concepts are to be discovered¹⁴. Consequently, questions like "Could x be an instance of y?" or "Could x be a sub-concept of y?" are not dealt with in a computational ontology. Furthermore, questions like "Is there an instance of x?" are not treated. The difference of these views also has its effect on the classification of new entities: In the intensional view, a new entity has properties and is to be assigned a matching concept. By contrast,

⁷ These graphs are different from the inheritance graphs as defined in (Horty, 1991), although inheritance graphs may be translated to concept graphs with attributes.

⁸ This is a postulation. A discussion of instances of multiple concepts or instances as concepts can be found in (Stein, 1991b).

⁹ For a definition of meta-concepts, see (Steele, 1990). For a concrete application of meta-concepts in programming, see (Kopp, 1996).

¹⁰ I use the notation $\exists!a: P(a)$ to state $\exists a: P(a) \wedge (\forall b: P(b) \Rightarrow a=b)$.

¹¹ Some frame modelling systems such as AROM (ROMANS project, 2003) allow so-called *multiple values* for one attribute.

¹² This view is highly disputable (cf. Eysenck and Keane, 2000).

¹³ Although this looks as if the first order frame was violated, this is not the case, as the relation is seen as an element of \mathfrak{R} and the predicate is \in .

¹⁴ This distinction is referred to as "prescriptive" versus "descriptive" concepts in (Padgham, 1991).

the extensional view demands naming the concept whenever a new entity is created. The entity then simply becomes an instance of this concept and obtains all attributes associated with it. Another consequence of the extensional view is that two entities or two concepts may be created, which share all attributes and all attribute values, but are different.

The distinction of instances and concepts has attracted much criticism (cf. e.g. Jones and Paton, 1998). The distinction might seem arbitrary, since every entity identifier could also be considered a concept which applies exactly and only to this entity. Furthermore, the uniform theory of instances and concepts ignores that some instances can be more typical prototypes of concepts than others can. It also ignores that concept membership may depend upon context or upon attribute values rather than upon a fixed hierarchy. Jones and Paton even argue that hierarchical concept organization is not adequate at all, since it cannot deal with atypical instances (1998). Although these objections are known, the computational model has proven to be extremely useful.

1.2.4. Objects, Classes and Fields

In order to model a domain with an object-oriented programming language, instances and concepts have to be translated into objects and classes. This section will introduce the equivalents of computational Ontology terms in the language of computer science. Often, two types of concepts are distinguished in computer science: Those basic ones, which are built-in in the programming language and those complex ones, which are defined by the programmer. The former are called *primitives*. They usually follow other rules of syntax and have no attributes. By contrast, the concepts defined by the programmer are called *classes*. The word *type* is a generic term for primitives and classes. In fact, object oriented programming consists of creating classes.

When a class is defined, its attributes are listed. Normal attributes are called *fields* in the terminology of computer science. Unlike static models, computational models also allow operations attached to classes. In the ontology, these would be attributes, the values of which are actions. Attributes of this kind are called *methods*¹⁵. Just like ordinary functions or procedures in computer science, methods may execute operations. They may have side effects and return a value. The instruction to execute a method is referred to as *calling a method*. A method call evaluates to the return value of the method. Although methods are entities in terms of the computational model, they are usually not part of the domain¹⁶. Just as concepts have super-concepts in the ontology, a class may have super-classes in a program. In the terminology of computer science, the sub-class is said to *extend* the super-class. Often, the analogy of a family tree is used: The super-class is called the *parent* of the sub-class. Accordingly, one can speak of *sister classes*, if two classes have a common parent class.

Programs may also create instances of classes. These are called *objects*. During the course of the program, objects may appear or vanish, values of their attributes may change and methods may be executed. These operations open doors to powerful means of modeling, which will be examined in this paper.

¹⁵ This concept differs from the idea of "procedural attachments" as discussed in (Winograd and Bobrow, 1977), because methods take up an ontological place, which is equivalent to the one of fields.

¹⁶ Some languages, among them CLOS and FAST, allow treating functions and methods as objects.

1.3. The Programming Languages CLOS, Java and FAST

This chapter will shortly present the programming languages CLOS, Java and FAST. I will broadly outline the history of these languages and explain why they have been chosen for this paper. Furthermore, I will mention their paradigms and their areas of application. In order to convey a feeling for the basic conceptualizations of CLOS, Java and FAST, the same sample program will be shown for each of them. Another section will address the background of object-orientation in Java, CLOS and FAST. Finally, the criteria for the comparison will be presented. They will help to determine the strengths and the weaknesses of the programming languages for the representation of ontological structures.

1.3.1. CLOS

In 1959, John McCarthy developed a programming language called *LISP*. LISP stands for *LIS*t *P*rocessor and is today one of the oldest programming languages still in use. Soon after its invention, many dialects of LISP evolved. In the eighties, a group around Guy L. Steele Jr. tried to unify the different dialects by developing a version called *Common LISP*. In 1990, an object-oriented extension for Common LISP was added. It is called *CLOS*, which means *Common LISP Object System*. The LISP community accepted Common LISP and CLOS and in 1994, the language was standardized by the ANSI.

Today, LISP is the most established language in the domain of Artificial Intelligence. It is widely used for speech recognition, robot control and planning algorithms. Since knowledge representation is a sub-domain of Artificial Intelligence, it seems promising to employ LISP in this field, too. If one wants to represent knowledge by objects, CLOS will be an obvious choice. Hence, this paper will explore in how far CLOS can be used for the modeling of ontological structures.

LISP belongs to the family of functional languages. It allows the manipulation of functions and lambda lists as first order elements of the language. What makes LISP different from other languages is its *code-as-data* paradigm: Every LISP program can be taken as a list of symbols and every adequate list of symbols can be evaluated as a LISP program. Since LISP is moreover interpreted or compiled incrementally, this paradigm allows for the runtime modification of the program as well as for the creation of new code at runtime. The equivalence of code and data has an immediate influence on the syntax of LISP: All LISP expressions have the form of a list, in which the function name is followed by the arguments.

Maybe it is due to this unusual syntactic taciturnity that LISP has often been deemed incomprehensible. Its proponents, however, are of the opinion that the lack of semicolons and syntactic structure frees the user's mind for the real issues in a program (Abelson and Sussman, qtd. in Steele and Gabriel, 1993). The liberal programming style is one of the most prominent features of LISP: It is a weak-typing language and there are no semantic checks done by the compiler. This has given LISP the label of being an "anarchic language", but has supported its widespread use for rapid prototyping. As the basic data-structure of LISP is the sequential list and as LISP is interpreted, its performance was often an object of criticism. However, powerful incremental compilers have been written and today, the performance of a LISP program can attain the one of a C program (Gat, 1999). Furthermore, the code-as-data paradigm of LISP opens doors to a huge number of dynamic facilities. In particular, LISP programs may analyze other LISP programs or create new ones. This predestines LISP for many applications in the domain of artificial intelligence. Unlike any other language invented in the fifties, LISP still enjoys constant popularity. This may partly result from its thorough theoretical foundations: LISP bases on the recursive function theory and the theory of computability. Moreover, the language has never stagnated: The fact that it can be extended and even changed so easily has constantly contributed to a spread of new ideas and versions.

A sample program can convey a first impression of LISP best. The purpose of the following line of LISP code is to print the sum of 9 and 23, along with a string:

```
`("The sum of 19 and 23 is" ,( + 19 23 ))
```

From a syntactical point of view, this line is a list of a string and an expression. The *quote* ` declares the list to be taken as a data list. This prevents LISP from interpreting the first element as a function name. The expression also has the form of a list: It is a function call for +, followed by its two arguments 19 and 23. Unlike the outer list, the inner one has to be evaluated as a mathematical expression. This is why the comma has been put. It functions as a *backquote* and undoes the effect of the preceding quote. If this line is entered in the LISP interpreter, it answers by displaying the list of "The sum of 19 and 23 is" along with the result of the sum.

1.3.2. Java

Java is one of the newest programming languages at this time. It first saw the light of day in 1990 as part of a software project of Sun Microsystems Inc. Initially, the group of James Gosling intended to use the programming language C++ for the project. As they encountered a number of problems with C++, they decided to create a new programming language – and Java was born (Gosling, 1995). Originally, the language was designed for embedded systems in the area of consumer electronics, but later Java matured to a rival of C++. When Netscape Inc. integrated Java in their browser in 1995, the language started conquering the Internet.

Java is often considered the state-of-the-art programming language in computer science. If one wants to represent knowledge by using a programming language, a computer scientist would maybe consider Java as a first choice. Furthermore, object-orientation forms an integral part of the language, which seems to predestine it for the modeling of ontological structures. This paper will explore in how far Java can compete with LISP, the classical language of artificial intelligence, for the purpose of object-oriented knowledge representation.

Java is entirely built upon the paradigm of object-orientation. Classes form the main structuring tool of the language. One of the primary goals of Java is program robustness. Hence, a great emphasis has been put on explicit declarations, typing and compile-time checks. Java source code is compiled to a special byte code, which is executed by a runtime environment. This concept ensures that Java programs can be executed on any platform and with any operating system – if the runtime environment is installed. In order to make the language look more familiar, Sun decided to take over nearly all syntactical elements of C. As a result, Java possesses a rich expressiveness of loops, declarations and operator expressions.

The call for explicitness in Java has been criticized as an "attitude of a governess" (Graham, 2001). On the other hand, it helps avoiding errors and supports the development of large and complex programs. As Java is an interpreted language, it is possible to do control at runtime, too. The runtime environment performs array index checks and cast checks, which makes Java programs considerably robust. As a drawback, this causes the performance of Java applications to be often behind the one of standard compiled languages. Java makes up for that by being truly platform independent. The language includes a huge number of pre-programmed classes (about 2700). They feature multithreading, event handling and graphical user interfaces. Hence, the programmer can fall back upon hundreds of sophisticated tools, which can be combined to new applications. Java also knows so-called *reflection classes*, which allow a program to analyze itself at runtime. Particular attention has been paid to the integration of the Internet: Java facilitates network programming and allows classes to be loaded dynamically from the web. Thus, Java programs may run with their components being distributed over the Internet. Furthermore, the fact that most Internet browsers have an integrated Java interpreter has given the language widespread popularity in the field of web-based applications.

The following sample program shall give a rough impression of the language:

```
public class Sum {
    public static void main(String[] argv) {
        System.out.println("The sum of 19 and 23 is " + (19+23));
    }
}
```

As every program has to be framed by a class statement in Java, the first line declares the class `sum`. Next, the function `main` is defined. It marks the start of execution. In order to display the string along with the sum, the object `out` is used. It represents the console and possesses the instance method

`println`. In terms of Java, the console is told to print the string. Since namespaces are separated in Java, the object `out` has to be accompanied by the class, which declares it, i.e. the class `System`.

1.3.3. FAST

FAST is a student's project. I wrote it from 1997 to 2003. Initially, the language was intended to be an extended assembler, which explains its name *Free and Straight Translator*. Later, ideas from many other languages influenced the project. FAST adopted elements from Pascal, C++, C, Java and LISP. Today FAST is a fully high-level stand-alone programming language.

FAST is a one-man project. It still has immature points and cannot be regarded an evenly matched rival of LISP or Java. Nevertheless, FAST was explicitly designed to support object orientation. Thus, it is in principle possible to use FAST for the modeling of ontological structures. This paper will investigate how FAST performs in this domain, as opposed to the established languages Java and CLOS.

FAST belongs to various language families. It combines aspects of object-oriented languages with ideas of functional languages as well as imperative and machine languages. The main goal of FAST is to form a coherent whole, which unifies the advantages of different programming languages. As a result, FAST offers a huge variety of programming tools, ranging from macros and machine instructions to classes and inline-functions. To keep the language simple in spite of that, a homogeneous syntax has been chosen, which resembles the one of BASIC. Since efficiency was a major aspect of the language design, FAST is a compiled language.

Similar to C, FAST relies on so-called include-files. These files define a great part of the language, comprising standard functions as well as control instructions such as `if` and `while`. When an include-file is needed, the FAST compiler integrates it automatically. This concept is intended to keep the compiler small and to encourage the expansion of the language. Writing the include-files in FAST is only possible because the 80386 Assembler Language is a subset of FAST. Consequently, little attention has been paid to platform independence. Up to now, FAST only compiles for MS-DOS. Furthermore, the current compiler produces a COM-file, which limits programs to a memory of 64 KB. These are significant drawbacks, which still remain to be fixed. FAST does justice to the word "free" in its name by allowing the programmer to compile nearly anything she or he writes (*coder-is-king* paradigm). In combination with a short code-and-run cycle, this is intended to simplify prototyping. On the other hand, the development of more complex programs is supported by a strong typing policy and object orientation. Furthermore, it is also possible to write FAST programs entirely in a functional style. Functions are equivalent data types, which may be stored in variables and be returned by other functions. Although FAST is a compiled language, it features dynamic memory management and garbage collection. This allows the programmer to concentrate on the programming task and ignore the problems of memory allocation.

The sample program in FAST clearly shows the heritage of imperative languages:

```
PRINT "The sum of 19 and 23 is "  
PRINTINT (19 + 23)
```

This program consists of a sequence of two instructions. Instructions are made up of a command, which is accompanied by one or more arguments. Instructions may have a result and they may be nested. In this case, the expression `(19 + 23)` is also an instruction, where the symbol `+` is the command and `19` and `23` are its arguments. This instruction is executed and its result is handed over to the command `PRINTINT`, which displays the sum on the screen. Technically speaking, the symbol `+` is an instance method of the class `integer` in FAST.

1.3.4. Classes in CLOS, Java and FAST

In order to provide the background for the following application of object-orientation, this section will shortly outline the abstract integration of classes in CLOS, Java and FAST. Java is the language in which classes are most conspicuous: Every Java-program consists of a set of files, each of which defines a class. These classes are seen as independent units, which are assembled to programs.

Thus, Java uses object-orientation to ensure modularity and reusability. As every program is necessarily a class, this may lead to degenerate classes for smaller programs such as the previous `sum` example. However, the technique has proven extremely useful for structuring larger programs. Although a class may act as a type, Java clearly distinguishes primitive types and classes in accordance with C.

In contrast to the design of Java, classes constitute just an additional feature in LISP: As LISP already had a long history, the language definition was not changed but rather extended. Nevertheless, CLOS bridges the gap between primitives and classes in an elegant way. Every class and every primitive is an instance of a so-called *meta-class*. Primitives are instances of `built-in-class`, whereas self-defined classes belong to `standard-class`. It is also possible to define new meta-classes, which can be used to adapt classes to special purposes. As the exact definition of meta-classes has not yet been approved by the ANSI, meta-classes will not be taken into consideration here. Since by default, every new class is automatically an instance of `standard-class`, this paper will assume all classes to be standard classes. The fact of classes being instances entails that classes can be handled as first order elements of the language. They may be stored in lists and be returned by functions. Furthermore, both primitives and classes may form hierarchies. Every class and every primitive is at the same time a type.

In FAST, classes are also integrated into the type concept. Two kinds of classes can be distinguished: so-called *Small Classes* and *Big Classes*. Both of them may form hierarchies and know inheritance. Small Classes are built upon an underlying primitive value. They implement primitives like `integer` or `boolean` and obey certain restrictions. By contrast, Big Classes may be used for complex concepts. Hence, the term *class* will only refer to Big Classes in this paper. In FAST, a class is seen as a function, which returns an instance. As functions are first order elements of the language, this also applies to classes.

In summary, both CLOS and FAST try to overcome the distinction between classes and primitives. Furthermore, they both allow access to classes as first order elements of the language. Java, by contrast, uses classes to structure program code and strictly distinguishes primitive types and classes.

1.3.5. Criteria for the Comparison

This paper will compare the representation of ontological structures in CLOS, Java and FAST. In order to do so, the criteria of comparison have to be fixed. A first criterion will be *powerfulness of design*. Strictly speaking, all languages can express every concept desired, as they are all Turing-complete. Nevertheless, there may be concepts, for which the language is not designed. These concepts can be expressed, but not in a reasonable way. An example would be multi-threading in BASIC: It is certainly possible to simulate multi-threading by alternating `goto`s, but one cannot say that the language was designed for it. The representation of ontological structures will border on limits of the design of the languages.

A second criterion will be the *expressiveness*, i.e. the ease with which a concept can be programmed. Although the design of a language may support a concept, this does not imply that the corresponding program is simple to write. Even more than powerfulness of design, expressiveness is subject to discussion. A first indication for expressiveness might be notational convenience (Bench-Capon, 1990). If a language allows implementing an idea with a less wordy program, then this language will be considered more expressive. Another aspect of expressiveness is uniformity. If a language allows expressing a concept in a coherent way and similarly to similar concepts, then I will judge this language expressive. A last aspect of expressiveness taken into consideration is *data abstraction*. This is the facility of a programming language to allow the use of a program component without knowledge of its implementation (Kopp, 1996). There are several other facets of expressiveness (cf. Bench-Capon, 1990), but as CLOS, Java and FAST mostly satisfy them, they will not be investigated here.

The last criterion for the comparison will be the *view of objects*. Some languages see objects merely as collections of fields, whereas others have a more complex idea of an object. In reality, entities are multifaceted beings, which act and interact in various ways. Hence, the more advanced the idea of an object is, the more adequate it is for Ontology. This criterion will soon show fundamental differences of the languages examined.

2. Basic Concepts of Object-Orientation

2.1. Classes and Attributes

The following chapters will show how concepts are represented in CLOS, Java and FAST. Although all three of them offer a highly sophisticated object-orientation, only the basic facilities will be analyzed and compared in this paper. The example used throughout the analysis is the following concept graph:

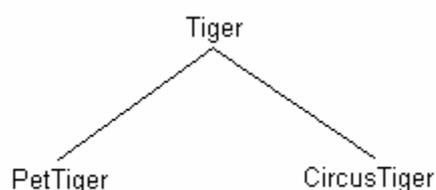


Figure 1: The sample concept graph

Tiger is the most general concept, of which **PetTiger** and **CircusTiger** are sub-concepts. This paper will start by defining the concept **Tiger** and later show how the two sub-concepts can be added to the model. It is assumed that tigers are in general dangerous, whereas pet tigers are not. In the last chapter, a special tiger will be introduced, which is both a **PetTiger** and a **CircusTiger**. It will be explored whether this one is dangerous or not.

This chapter will investigate how concepts and attributes can be translated to program code. After having declared the class **Tiger** with a normal field, two special kinds of fields will be examined: Those, which share the same value for all instances and those, which may not be accessed from outside.

2.1.1. Classes

All three programming languages examined require the definition of a class before an object can be created. At first, the class **Tiger** will be just equipped by one attribute: its name. By declaring this attribute, it is stated that it does make sense to have a tiger with a name – which distinguishes tigers for instance from abstract entities, which usually do not bear names. The class definitions in CLOS, Java and FAST are as follows¹⁷:

CLOS	Java	FAST
<pre>(defclass Tiger () ((name)))</pre>	<pre>class Tiger { String name; }</pre>	<pre>CLASS Tiger (VAR name "")</pre>

These declarations differ primarily by their syntactic style: CLOS requires the class definition to be a list, surrounded by round brackets. **defclass** is a macro, which expects as arguments the name of the class (**Tiger**), a list of super-classes (none) and a list of attributes, each of which is again a list (**(name)**). By contrast, Java uses braces and semicolons in accordance with C. Furthermore, Java requires the attribute to have a type (**string**). The FAST-declaration is very similar to the Java program, although it lacks braces and semicolons. Moreover, FAST requires all field declarations to provide an initial value (the empty string in this case)¹⁸. The initial value implicitly determines the type of the field. FAST requires typing by design, as it is a compiled language with the goal of efficiency. As a side effect, this policy ensures that every field has a value and avoids **null**-pointer exceptions.

¹⁷ FAST and CLOS are not case-sensitive. This paper uses the language-dependent conventions of capitalization.

¹⁸ Initial values are not necessarily purposeful default values in the ontological sense. See (Bench-Capon, 1990) for a criticism of omnipresent default values.

In object-orientation, a strong typing policy is commonly regarded an advantage. In the above example, FAST and Java both guarantee that the field `name` will only contain strings. This serves as a minimal semantic check for the field value. In CLOS, by contrast, values are typed, but the fields that contain these values are not. This calls for a high degree of discipline, since LISP distinguishes for instance between strings and symbols. If the programmer decides to represent the value of `name` by a symbol and later compares the `name` to a string with the same sequence of characters, then this comparison will simply evaluate to `NIL`¹⁹.

The above classes differ by another characteristic, which is their position in the concept graph. CLOS and Java know a *mother class* of all classes. Every class, including `Tiger`, is automatically direct or indirect sub-class of this mother class. In CLOS, this class is called the `standard-object` and in Java, the class is `Object`. This causes the concept graph to have a uniform, tree-like structure with one root. Technically speaking

$$\exists!x: \forall y: \text{concept}(y) \Rightarrow \text{subconcept}(y,x)$$

This design is useful if objects of different classes are to be treated alike, such as for instance in an array. FAST also provides a class `Object`, which is intended to fill this role of a mother class. However, due to its coder-is-king paradigm, FAST cannot force the programmer to accept `Object` as a super-class. As a result, the concept graph of FAST may fall into non-interconnected sub-graphs, if the programmer desires so. The existence or absence of a forced mother class can be considered the main difference of class declarations in CLOS, Java and FAST.

2.1.2. Shared Attributes

The attribute `name` defined so far will have different values for different tigers. However, there may be attributes, which share the same value for all direct instances of a class. For the example, it will be assumed that all tigers are equally dangerous. In terms of the computational ontology, the restriction of the relation `dangerous` to tigers has a singleton co-domain. Attributes of this kind will be called *shared attributes*. In this case, the concept itself is said to have that value for the attribute:

$$\forall R,c: \text{concept}(c) \wedge (\exists!y: \forall x: \text{direct-instance}(x,c) \Rightarrow (x,y) \in R) \\ \Rightarrow \exists!y: (c,y) \in R \wedge (\forall x: \text{direct-instance}(x,c) \Rightarrow (x,y) \in R)$$

Note that this does not mean that a shared attribute describes a concept. Being dangerous is not a property of the concept `Tiger`. It is a property of all instances of `Tiger`. Neither does this mean that a shared attribute describes a collective property of the instances. The property of being dangerous does not emerge from the ensemble of tigers, but it is inherent to each of them. The notion of shared attributes abbreviates the fact that all direct instances of a concept have a common value for it.

In programming languages, this boils down to the fact that the attribute is just stored once per class. In the example, it will be supposed that all tigers are dangerous to a degree of 100%. To model this in CLOS, Java and FAST, the following line has to be added within the class declaration:

CLOS	Java	FAST
<code>(dangerous :allocation :class :initform 100)</code>	<code>static int dangerous=100;</code>	<code>SHARED dangerous 100</code>

These declarations differ by syntax, but not by semantics: CLOS defines the properties of attributes by so-called *slot-specifiers*. In the above example, `:allocation` is a slot-specifier, the value of which is `:class`. This means that `dangerous` is a shared attribute, which exists only once per class. Furthermore, the slot-specifier `:initform` is used to assign a value to the field, namely 100.

¹⁹ Although LISP provides typing by the slot-specifier `:type`, a type violation does not result in an error, but in an "undefined" behavior (Steele, 1990).

Java uses the modifier `static` to indicate that an attribute is shared. Value assignments can be done simply by an equation sign. As Java always requires typing, the word `int` declares the type of the field to be integer. FAST uses a similar approach: By the use of `SHARED` instead of `VAR`, the attribute becomes shared. Although shared attributes need a constant initial value given, they may vary during the course of the program. Again, typing is implicit in FAST.

As the declaration of a shared attribute is possible in CLOS, Java and FAST, no difference in powerfulness of design can be stated. Concerning the expressiveness, it can be observed that CLOS requires a much more wordy declaration than Java and FAST.

2.1.3. Private Attributes

In a computational model, it might be useful to hide some attributes of an object. This applies mostly to internal states of instances, which are not to be seen from outside. In the example of a tiger, the state of being hungry would be an attribute, which can neither be accessed nor changed except by the tiger itself. Such an attribute will be called a *private attribute*. In object-oriented programming, private attributes also serve another purpose: If a field cannot be accessed from outside, this enables the object to keep up an internal coherence. This policy is called *encapsulation*. In the example, the state of being hungry could be linked for instance to an action `eat`, such that the tiger is replete if and only if the action `eat` was performed. In order to declare a private attribute `hungry`, the following lines have to be added within the class declaration of `Tiger`:

CLOS	Java	FAST
<code>(hungry)</code>	<code>protected boolean hungry;</code>	<code>VAR _hungry false</code>

Concerning the concept of privacy, the languages differ significantly: CLOS, as the oldest language, does not support private attributes at all. It is not possible to prevent the access to an attribute from outside²⁰. Unlike CLOS, FAST has a weak concept of privacy: If an identifier starts with an underbar, it is by convention to be regarded as a private attribute. This applies to methods as well as to fields. If the attribute is accessed from outside, FAST displays a warning. In spite of that, it does not abort the compilation process. This policy is a trade-off between the desire to support privacy and the *coder-is-king* paradigm, which does not allow the compiler to restrict the programmer. It is assumed that clean programs should not produce warning messages.

By contrast, Java has a very powerful visibility regulation. It knows four modifiers, which specify where an attribute may be accessed. The most general one is `public`: It allows the attribute to be seen everywhere. The modifier used above (`protected`) restricts the access to the class and all of its sub-classes. Whenever program code from outside tries to use the identifier `hungry` in context with an instance of `Tiger`, Java displays an error and aborts the compilation. Thus, it is impossible to manipulate the field by non-authorized code. If the modifier `private` is used, the access is limited to the class itself. No other class may retrieve the attribute value or change it. This modifier also serves optimization purposes, as private fields need not be transferred to sub-classes, because they cannot be accessed anyway²¹. If no modifier is provided, a so-called *default modifier* is assumed, which restricts access to all sub-classes and all classes in the same unit. These units are called *packages* and group related classes in a directory. The application of modifiers also extends to methods and even classes. The strong policy of visibility results from the goal of robustness in Java. By protecting attributes from external influence, Java code can guarantee the coherence of the state of an object. In the terminology of Java, the access restrictions provided by FAST are `protected` and `public`, whereas CLOS only knows `public`.

2.1.4. Chapter Summary

In this chapter, a concept with its attributes has been translated to a class. All languages allow expressing concepts and fields in a similar way. The main difference between the languages is their

²⁰ Although CLOS supports accessor-methods (s.b.), this does not forbid the access to the field by means of `slot-value` (Steele, 1990).

²¹ An inner class, which inherits from the outer one, may access the private fields in spite of that.

idea of objects. The concept of visibility in Java (and, though less strict, in FAST) may indicate that both languages have a view of objects, which is more complex than the one of CLOS. In Java and FAST, objects are seen as units, which have a certain control over their state. In CLOS, by contrast, objects are rather considered tuples, which can be manipulated arbitrarily by the programmer. Thus, Java and FAST share a more powerful view of objects.

2.2. Methods

This chapter will analyze how abilities of instances can be integrated into the model. In the computational model, a capability corresponds to an attribute, the value of which is an action. These action attributes, called *methods*, allow leaving the frame of the static model. Instances may act and interact, make use of the screen, of the hard drive and even of the Internet. In the programming languages analyzed here, a method is very similar to a normal function: A method may receive arguments, have side effects and return a result. As these mechanisms belong to the standard repertoire of a programming language, they will not be discussed here. Rather, the integration of functions to object-orientation will be examined in detail: First, a normal action attribute will be declared. Then, two special scenarios will be envisaged, which make greater demands to the languages. Last, it will be shown how methods can be used in combination with private attributes.

2.2.1. Normal Methods

Entities may have certain abilities. Tigers for instance can roar. Java, CLOS and FAST implement instance abilities by methods. In the example, it will be assumed that all tigers roar the same way, namely by displaying a string on the screen:

CLOS	Java	FAST
<pre>(defmethod roar ((a Tiger)) (print "Rooaar!")))</pre>	<pre>void roar() { System.out.println ("Rooaar!"); }</pre>	<pre>FUNCTION roar (PRINT "Rooaar!")</pre>

Concerning method declarations, Java and FAST differ significantly from CLOS. Both languages require a method to be defined inside the class. Fields and methods are treated alike within a class: In Java, the declaration of a method differs from the one of a field just by the method body. The return type of the method takes the place usually occupied by the type of a field. As the above method does not return any value, its return type is `void`. In FAST, methods are declared by using the word `FUNCTION` instead of `VAR`. Just like in Java, the body of the method follows its name. The return type of a method is implicit in FAST. Both languages hand an implicit argument to the methods, which is called `this`. This argument refers to the object, to which the method is attached. In the above example, none of the methods makes use of `this`. To sum up, Java and FAST share the idea of an attribute, the value of which is an action.

By contrast, CLOS requires the method definition to be outside the class. The method is associated to the class by declaring a typed argument. Here, the argument is `a`, the type of which is `Tiger`. Now, `a` has a function similar to `this` in Java and FAST: It refers to the object, to which the method is attached. In this case, a CLOS method behaves like a Java method or a FAST method. However, it does not exactly share the idea of an action attribute, because it is defined outside the class. The method is rather seen as an independent entity, which is loosely related to a class. This design allows adding methods to the class after the class has been declared, which is not possible in Java or FAST²². As I will show in the next section, the method declarations in CLOS have another significant advantage.

2.2.2. Multi-Methods

The action defined so far depended only on one class, namely the class `Tiger`. I will now examine the case, in which an action is carried out in dependence on more than one object. For the example, it will be assumed that different things happen when tigers meet other tigers and when tigers meet antelopes. In order to model this, a class `Antelope` is taken for granted. The program in CLOS would look as follows:

²² Although Java knows overloading and FAST knows external methods, these are not equivalent to normal methods. They cannot be launched in dependence on the run-time type of the argument.

```

(defgeneric meet (a b))

(defmethod meet ( (a Tiger) (b Tiger) )
  (print "Tiger meets tiger")
)

(defmethod meet ( (a Tiger) (b Antelope) )
  (print "Tiger eats antelope")
)

```

The first line declares a so-called *generic function*. It serves as a template, which states that any method called `meet` requires two arguments. The following two declarations define a method for the meeting of two tigers and the meeting of a tiger and an antelope, respectively. Whenever `meet` is called with two tigers, the first method will run, whereas the meeting of a tiger and an antelope will launch the second method. This dynamic search for an applicable method is called *generic method dispatch* in CLOS. The relation of the methods to the classes is solely given by the typing of their arguments. If a method is associated to more than one class, then it is called a *multi-method*. If the generic function is not given (as it was the case for `roar`), it is created implicitly.

CLOS offers a number of additional facilities for the configuration of a generic function. In particular, generic functions are instances of meta-classes, which determine the behavior of the functions. The programmer may for instance specify the order in which the methods are checked for applicability. This can be useful in case of an under-specification, which would allow several methods to be called. Although the feature of multi-methods is a clear proof of the powerfulness of design in CLOS, it has to be kept in mind that a generic method dispatch involves a search for the appropriate method at run-time.

The concept of generic functions is lacking in Java and FAST. In order to model the above action, the method `meet` has to be defined within a class. This already enforces a design decision, because neither the class `Tiger` nor the class `Antelope` are a priori predestined for the meeting of a tiger and an antelope. I assume that the method is declared in the class `Tiger`. Apart from the implicit argument `this`, the method has another argument, which is of the most general type `Object`²³. This argument will receive either a tiger or an antelope. It is supposed that `Tiger` is a sub-class of `object` in FAST²⁴. The method body has to check whether the argument is a tiger or an antelope and perform the appropriate action²⁵.

Java	FAST
<pre> void meet(Object a) { if(a instanceof Tiger) System.out.println ("Tiger meets tiger"); if(a instanceof Antelope) System.out.println ("Tiger eats antelope"); } </pre>	<pre> FUNCTION meet a object (IF (isa a Tiger) (PRINT "Tiger meets tiger") IF (isa a Antelope) (PRINT "Tiger eats antelope")) </pre>

In summary, Java and FAST allow the modeling of the above situation, but not by design. Hence, CLOS possesses a greater powerfulness of design in this issue. Since generic functions have no pendant in Java and FAST, their analysis will not be deepened. Furthermore, the idea of methods as action attributes will be kept up, as it provides a useful model for most methods.

²³ Although Java allows method overloading, it does not support a generic method dispatch, which depends on the run-time type of the argument. If two different methods for antelopes and tigers were declared, Java would not know which method to call if the animal has been cast to `object`.

²⁴ Super-class declarations will be treated later in this paper. For purposes of demonstrations, the super-class is taken for granted here.

²⁵ Strictly speaking, the function `Tiger` should be quoted in FAST by `LAMBDA`. As `isa` is a macro, this is not necessary.

2.2.3. Variable Instance Methods

Up to now, the actions defined for a class always concerned all of its instances. All tigers roared the same way. In terms of the computational model, `roar` was a shared attribute. It should be possible to have tigers, which roar in different ways, just as it is possible to have tigers, which carry different names. Methods of this kind will be called *variable instance methods*. Although this seems to be a simple demand, only CLOS and FAST offer simple solutions.

In CLOS, methods may not just specialize on classes, but also on instances. In order to declare a method, which is attached to an instance, the type specifier `eq1` (equal) is used. Presupposing a tiger Bob²⁶, the following declaration would make Bob's roaring more impressive:

```
(defmethod roar ( (a (eq1 bob)) )
  (print "Rooooooooaaaaaar!")
)
```

The place of an argument type is taken by the list `(eq1 bob)`. The generic dispatch is configured to prefer methods, which specify an `eq1`-argument. Thus, the above method would be preferred if `roar` was called for Bob, although the standard `roar` for tigers would also be applicable. Hence, CLOS provides a convenient way for defining methods attached to instances. It has to be kept in mind, however, that the call of a generic function involves a search for the applicable methods and a precedence sorting at run-time.

In order to write the corresponding program in FAST, it is useful to know that the declaration of the standard `roar` function can be written in two equivalent ways:

<pre>FUNCTION roar (PRINT "Rooaar!")</pre>	<pre>CONST roar (LAMBDA (PRINT "Rooaar!")</pre>
--	---

The use of `FUNCTION` is an abbreviation for declaring a constant attribute, the value of which is a function. In the right code fragment, the attribute `roar` becomes a function, because its initial value is a `LAMBDA` expression. Furthermore, the attribute is constant, i.e. its value may not be changed in the course of the program.

If the word `CONST` is now exchanged by `VAR`, then the value of the attribute is allowed to change. As a result, it is possible to assign new functions to `roar`. Thus, individual tigers may have individual implementations of the method. This is facilitated by the fact that FAST supports functions as first order elements of the language. Hence, functional values are treated similar to string literals or integer values in a program. In order to change Bob's roaring, the following line would suffice:

```
LET (LAMBDA roar bob) (LAMBDA this Tiger (PRINT "Rooooooooaaaaaar!"))
```

The first expression identifies the roaring method of Bob. The word `LAMBDA` is used to avoid the method to be called²⁷. The second expression provides a new functional value. For reasons of type congruence, the argument `this` has to be specified explicitly. After this assignment, the object Bob has a different implementation of the method `roar`. By regarding methods as attributes with a functional value, FAST overcomes the strict distinction of methods and fields: Methods may change like fields. Unlike CLOS, FAST can call such a method without a dynamical search process²⁸.

Java does not allow methods, which are attached to instances. In order to program the above scenario with Java, a new class has to be created, which contains a template for the `roar` method. Then, each individual `roar` function has to be implemented in a sub-class. Such a program would involve inner classes, inheritance and method overwriting. As these advanced concepts have not yet been

²⁶ Instance creation is treated later in this paper. For purposes of demonstration, the instance is taken for granted here.

²⁷ The use of `LAMBDA` corresponds to a function quoting in LISP. The accessor functions introduced below may be used instead.

²⁸ On the implementation level, this translates into a normal instance memory cell, which stores the function address.

introduced, it will be abstained from a sample program. One can say that the design of Java does not support instance methods as this is done in CLOS or FAST. As a result, Java lacks powerfulness of design for this issue.

2.2.4. Accessor Methods

Up to now, private attributes could not be accessed at all. If the attribute `hungry` is declared `protected` in Java or starts with an underbar in FAST, then it is neither possible to retrieve nor to change its value from outside without producing an error or a warning. In certain cases, it might be useful to allow retrieval, but not a change of the value. In the example, it might be imagined that it is possible to see whether a tiger is hungry, but not to change this state. All three languages examined support this concept by the use of so-called *accessor methods*. There may be two accessor methods associated to an attribute: A *reader method* and a *writer method*. The reader method is called to retrieve the attribute value, whereas the writer method is called to change it. When a private attribute possesses a reader method, but no writer method, then this corresponds to the case where the attribute may be retrieved, but not changed. For purposes of demonstration, the attribute `hungry` will be re-declared with both accessor methods in CLOS, Java and FAST:

CLOS	Java	FAST
<code>(hungry :accessor hungry)</code>	<pre>private boolean hungry; boolean getHungry() { return hungry; } void setHungry(boolean h) { hungry=h; }</pre>	<code>VAR hungry false</code>

As shown here, Java is the only language, which requires explicit accessor methods. The methods are by convention named `getField` and `setField`. The reader method just returns the field, whereas the writer method receives the new value of the field as an argument and assigns it to the attribute. Presuming a tiger Bill, retrieving and setting the field value would look as follows:

```
bill.getHungry()
bill.setHungry(true)
```

The issue is handled differently in CLOS: If the slot-specifier `:accessor` is given, CLOS creates the appropriate accessor methods on its own. The reader method carries the name following the specifier, while the writer method has a more complicated name: It is a list, which contains the symbol `setf` and the given name. In the example, calls to the accessor methods are

```
(hungry bill)
((setf hungry) t bill)
```

The first line is a normal function call to the reader method. The second line is a call to the writer method, where the list `(setf hungry)` is the method name. The first argument is the new value (`t` for `true`), and the second is the object. CLOS also allows specifying reader and writer methods separately. Similar to `:accessor`, the slot-specifiers `:reader` and `:writer` may be used. Each of them requires a name given. The first declares a reader method and the second a writer method. Thus, CLOS provides a sophisticated accessor policy, which can make up for the lack of private attributes. However, as Sowa pointed out, the use of accessor methods for encapsulation results in an inefficient overhead of real function calls for a syntactical feature (2000). Each access to the field entails a costly generic function dispatch (Cyphers, 1990). Moreover, methods can never be declared private.

In FAST, reader and writer methods are created automatically for non-private attributes. In contrast to CLOS and Java, the methods are only created syntactically. Although they may be used like normal

methods, they do neither create code nor provoke a function call, which complies with the goal of efficiency. In FAST, an access to the field `hungry` would look as follows:

```
(hungry bill)
LEThungry bill true
```

The first line equals the method call used in CLOS. It retrieves the value of the field. The second is a call to the method `LEThungry`, which receives the object and the new value as arguments. The advantage of this approach is a gain in data abstraction: It is not necessary to know whether a field has accessor methods or not. The methods can always be used, without a loss of efficiency. If the programmer later decides to make the attribute private and to hand-code the accessor methods, none of the existing code needs to be changed. In summary, CLOS and FAST provide a rather mature concept of accessor methods, whereas Java relies on explicit methods. It is easier to define accessor methods in CLOS and FAST, which questions the expressiveness of Java in this issue.

2.2.5. Chapter Summary

In this chapter, different kinds of methods have been declared. It became apparent that CLOS is able to express normal methods as well as multi-methods and variable instance methods. FAST can express normal methods and variable instance methods, but no multi-methods. Java can only express normal methods. Thus, a clear decrease in powerfulness of design can be shown from CLOS to Java in the issue of action attributes.

In the last section, accessor methods were investigated. It was shown that CLOS and FAST provide a more sophisticated support for accessor methods than Java. This results in a higher expressiveness of CLOS and FAST in the issue of private attribute accesses.

2.3. Object Creation and Attribute Access

This chapter will show how instances of classes can be created in CLOS, Java and FAST. Although this is one of the most important and most often used processes in programming languages, the dynamic instance creation does not have a counterpart in the classical static first order logic description of Ontology. In the example, a tiger Shirkan will be brought into being and it will be demonstrated how Shirkan receives his name. This will again stress the difference in the view of objects in CLOS, Java and FAST. Furthermore, it will be examined how the attributes of Shirkan can be accessed. This analysis will be evidence for further strengths and weaknesses of the languages.

2.3.1. Object Creation

All programming languages examined create instances of classes by use of a special method, called the *constructor*. The constructor allocates memory and initializes the object on an internal level. Usually, the fields of the object are cleared or marked as unbound and a reference to class information is established. Note that the constructor cannot be a normal method, as it has to be called in absence of an instance. For CLOS, Java and FAST, the creation of a tiger Shirkan would look as follows²⁹:

CLOS	Java	FAST
<pre>(setq shirkan (make-instance 'Tiger))</pre>	<pre>Tiger shirkan= new Tiger();</pre>	<pre>VAR shirkan Tiger</pre>

In CLOS, the constructor is provided by the system. It is called `make-instance`. In order to create Shirkan, the constructor is called with the name of the class, `Tiger`. The name is preceded by a quote `'` in order to avoid the symbol `Tiger` to be considered a variable. The result of the constructor call is assigned to a variable named `shirkan`.

In Java, the constructor is also provided by the system. It bears the name of the class, `Tiger`. In tradition of C++, constructors are called with the reserved word `new`, which allocates memory. The above line declares a variable `shirkan` of type `Tiger`, calls the constructor and assigns its result to the variable.

Although the constructor call looks similar in FAST, the underlying structure is different. In FAST, the class is itself the constructor. A class is a function, which returns an instance. In the above program line, the constructor is called and its result is assigned to the variable `shirkan`. Implicitly, `shirkan` receives the type `Tiger`.

Up to now, all three languages behave similarly for the creation of an object. There is no considerable difference in expressiveness. It may be noted, however, that the terminology of CLOS stresses the rather technical view of objects. Whereas Java and FAST only require the concept to be named, CLOS expects a call to the function `make-instance`. Thus, the creation of an object is more technical and less abstract in CLOS³⁰.

2.3.2. Object Initialization

Another task of a constructor is assigning initial values to the fields of the new instance. In the example, it would be useful if the constructor call for `Tiger` could immediately provide a `name` for the new instance. To do so, the class definition needs to be modified in all three languages:

²⁹ It is assumed that the declarations are local to functions, so that the asterisks are not used in CLOS and the word `static` can be omitted in Java.

³⁰ A criticism of `make-instance` can be found in (Keene, 1989).

CLOS	Java	FAST
<pre>(defclass Tiger () ((name :initarg :name) (dangerous :allocation :class :initform 100))</pre>	<pre>class Tiger { String name; static int dangerous=100; Tiger(String n) { name=n; } }</pre>	<pre>CLASS Tiger n string (VAR name n SHARED dangerous 100)</pre>

CLOS differs from Java and FAST by associating fields directly to constructor arguments. In the above program, the slot-specifier `:initarg` declares an argument called `:name`. Now, the following line of code may create Shirkan and initialize the field `name`:

```
(setq Shirkan (make-instance 'Tiger :name "Shirkan"))
```

Java follows another approach: Although the constructor can be created automatically, it may also be defined manually. It is a special method, which bears the name of the class and does not have a return type. In the example, the constructor takes an argument `n`, which it assigns to the `name` of the newly created instance. For Shirkan, the constructor call looks as follows:

```
Tiger shirkan=new Tiger("Shirkan");
```

FAST is similar to Java, although no constructor needs to be defined, as the class is itself the constructor. Thus, the class declaration itself takes the arguments. An assignment of the argument `n` to the field `name` is then particularly simple, because `n` can be given as an initial value to `name`. If a function call has arguments, it needs to be surrounded by brackets and so Shirkan would be created by

```
VAR shirkan (Tiger "Shirkan")
```

The above example shows that Java requires more code to assign initial values than CLOS. In Java, it is necessary to define the constructor by hand. On the other hand, the concept of a constructor is more powerful than the idea of initializing arguments in CLOS. Firstly, it is not restricted to filling up the instance fields. Constructors in Java may perform extensive initialization tasks, call methods, display debug messages or even prevent the creation of the instance. In the example, one could choose to up-case the tiger's name before assigning it. The `make-instance` function of CLOS is not intended for these purposes. Secondly, the constructor of Java hides the implementation details of the class. In CLOS, the call to `make-instance` reveals the fields of the class. Whenever the internal structure of the class changes, it is necessary to rewrite all code that executes calls to the constructor. This results in a loss of data abstraction. Although it is possible to simulate the constructors of Java by replacing certain built-in methods, this is rarely done. It is in general suggested to create a normal function like `make-tiger`, which calls `make-instance` and then carries out initialization tasks³¹.

FAST unifies both approaches: It is possible to write constructor code inside the class, but it is at the same time possible to restrict oneself to assigning constructor arguments to fields. As these facilities are offered without the necessity of a new language concept, FAST may be considered more expressive in this issue. For object-oriented knowledge representation, the above discussion shows again that Java and FAST have a more complex idea of an object: It is an independent whole, which can carry out actions when it is created. It is the task of the object itself to care for initial field values. CLOS, by contrast, sees the object as a tuple, which is filled up by the constructor.

2.3.3. Access to Attributes

After having created a tiger, the most important question is how to access its fields. This is one of the most basic tasks for an object-oriented programming language. In CLOS, Java and FAST, the `name` of Shirkan can be retrieved as follows:

³¹ This idea is for instance supported by (Keene, 1989) and (Slade, 1998).

CLOS	Java	FAST
<code>(slot-value shirkan 'name)</code>	<code>shirkan.name</code>	<code>(name shirkan)</code>

In CLOS, the function `slot-value` is used to retrieve the value of a field. Firstly, this entails a more complicated way of accessing attributes than the approach used in Java and FAST. Both of them do not need a function call. Secondly, it shows again the more technical view of an object. While Java and FAST only require naming the object and the attribute, CLOS reminds the programmer of the view of an object as a tuple of slots. If accessor methods are used, the problem vanishes: A retrieval of a field value then looks exactly like in FAST. Nevertheless, the existence of `slot-value` is again a proof of the more technical standpoint of CLOS.

If an accessor method is used, CLOS gives a functional shape to the retrieval of a field value. The object follows the attribute name. The attribute is the head of the expression. This syntactical style goes along with the functional nature of CLOS. By contrast, Java has the object name precede the attribute name. In the view of Java, the programmer holds the object and asks it for its attribute value. CLOS focuses on the attribute, whereas Java focuses on the object. If one is willing to abstract, one can see parallels of these different philosophies in many other domains. The general question is whether to focus on the goal or the object. In this terminology, the *goal* is an attribute, an action or information, while the *object* is the target of this operation. The different foci can for instance be found in the different operating system paradigms: Command-oriented operating systems have the command (i.e. the goal) precede the argument (i.e. the object). By contrast, window-oriented operating systems allow choosing the icon (i.e. the object) before selecting the action (i.e. the goal). Another example is linguistics, where the head and the complement of a noun phrase reflect the goal and the object of its meaning. In some languages, the goal precedes the object (cf. French "Le nom du tigre"), whereas in others, the object precedes the goal (cf. German "Der Tigername"). In mathematics, some operators follow their argument (cf. `1!`), whereas others precede it (cf. `sin 1`). In summary, there is no canonic preference for a focus on the object or a focus on the goal.

In FAST, the focus is in general on the goal, i.e. the attribute name precedes the object name. However, if the attribute name is up-cased, it may also follow the object. This allows a focus on the object:

```
(name shirkan)
(shirkan NAME)
```

FAST is not case-sensitive. The capitalization is just required to maintain the readability of the program. Independent of this, the compiler can always resolve an attribute name in the second place unambiguously. This syntactical feature explains why arithmetic expressions may be written infix in FAST: Arithmetic operators are instance methods of the class `integer`. They may follow their objects, because they do not contain any lower-case characters (Suchanek, 2003). Thus, FAST unifies infix and prefix notation. It allows both a functional and an object-oriented viewpoint. It offers more flexibility than Java and CLOS in the issue of attribute access.

2.3.4. Access to Shared Attributes

Each `Tiger` has a different value for the field `name`. Thus, it only makes sense to ask for a name if a certain Tiger is given. By contrast, it should be possible to retrieve the value of a shared field in absence of an instance, too. In the example, it should be possible to find out whether tigers are dangerous without actually meeting a tiger. It turns out that this is only allowed in Java and FAST:

CLOS	Java	FAST
	<code>Tiger.dangerous</code>	<code>(dangerous Tiger)</code>

Java and FAST can treat the class name like an object. As a result, shared fields may be accessed similarly to normal fields. In order to achieve this in CLOS, `Tiger.dangerous` would have to be declared as a global variable outside the class. As this leaves the frame of object-orientation, a subclass, such as `CircusTiger`, would not inherit the attribute. Hence, this alternative cannot be

regarded a valuable solution. The self-inspecting capabilities of CLOS would also allow retrieving the value of `dangerous`, but there is no simple predefined way to access a shared field without creating an object. Thus, CLOS lacks powerfulness of design in this issue.

2.3.5. Method Calls

This section will illustrate how methods are called in CLOS, Java and FAST. In the example, the tiger Shirkan will be told to roar. In object-oriented languages, launching a method is called *sending a message to an object*. The message `roar` is sent to the tiger Shirkan. Object-oriented programming means having objects send messages to other objects. In CLOS, Java and FAST, a call to `roar` looks as follows:

CLOS	Java	FAST
<code>(roar shirkan)</code>	<code>shirkan.roar()</code>	<code>(roar shirkan)</code>

All languages express method calls similarly to field value retrieval. This idea is perfected in FAST: Even without accessor methods, there is no difference between the use of a method and the use of a field. This proves a high degree of data abstraction: An attribute may be declared either as a method or as a field, depending on whether the result has to be calculated or has been stored explicitly. This is an implementation detail, which shall have no influence on the use of the attribute. It is not possible to hide this implementation decision from the user without creating accessor methods in CLOS and Java. In FAST, the programmer does not have to know whether the value of an attribute is created dynamically or has been stored statically. Thus, FAST is more expressive in this issue.

Another implementation detail is whether the function has been declared as a method or as a normal procedure. In order to guarantee data abstraction, Kopp demands a uniformity of method calls and function calls: If a normal procedure takes an object for its first argument, a call to it should not differ from a call to a method (1996). CLOS and FAST support this view: A function call is always a function name followed by its arguments. It is not necessary to know where the function has been declared in order to use it. Java, by contrast, distinguishes normal functions and methods strictly. Hence, both CLOS and FAST are more coherent and thus more expressive than Java in this issue.

2.3.6. Chapter Summary

In this chapter, an object was created and its attributes were accessed. The implementations of these basic tasks showed again that CLOS has a more primitive idea of objects than Java and FAST. The use of terms like `make-instance` and `slot-value` proved the more technical view of objects in CLOS. Furthermore, the lack of a complex constructor indicates that objects are seen as tuples of slots rather than self-responsible entities. Moreover, CLOS is not able to access shared attributes without an object. Both Java and FAST proved to have a more powerful design in this issue.

Throughout the chapter, slight advantages of expressiveness became apparent in FAST. First, constructors need not be defined explicitly. Second, attributes may precede or follow the object, depending on the programmer's view. Third, methods and fields follow the same uniform usage, resulting in a high degree of data abstraction. Last, methods and functions also follow the same uniform usage, allowing the programmer to ignore implementation details. In general, it can be said that methods integrate more coherently into CLOS and FAST than into Java.

3. Inheritance

3.1. Single Inheritance

This chapter will introduce the concept of pet tigers. Pet tigers are tamer and less dangerous than normal tigers; they are a special kind of tigers. It will be analyzed how this specialization finds an expression on the ontological level and how it can be translated to a program. Last, the classical inference problem of Ontology will be presented and solved in CLOS, Java and FAST. It will be shown that Java lacks powerfulness of design for shared attributes.

3.1.1. Sub-Classes

As every pet tiger is a tiger, `PetTiger` must be a sub-concept of `Tiger`. Although by definition, sub-concepts share all attributes of their super-concept, they may differ in various ways from their super-concepts. In particular, they may have additional attributes. Even if a sub-concept has more attributes than its super-concept, it still satisfies the `subconcept` relation. In the example, a pet tiger differs from a normal tiger by living with a human. This relation can be modelled by an attribute called `owner`. Despite of this additional attribute, every pet tiger is still a tiger. It is just more "powerful" in the sense that it has more properties. Note that by definition, a sub-concept may never have fewer attributes than its super-concept. If a pet tiger for instance did not have a name, then a pet tiger could not be a tiger, because every tiger has a name.

In CLOS, Java and FAST, a sub-concept translates to a sub-class. In the example, a class `PetTiger` is defined, which extends the class `Tiger` and adds the attribute `owner`:

CLOS	Java
<pre>(defclass PetTiger (Tiger) ((owner)))</pre>	<pre>class PetTiger extends Tiger { String owner; PetTiger(String n) { super(n); } }</pre>
FAST	
<pre>CLASS PetTiger n String (EXTENDS Tiger n VAR owner "")</pre>	

CLOS uses a list style for specifying the super-class `Tiger`, whereas Java and FAST use the keyword `extends`. All three languages allow declaring a sub-class without repeating the attributes of the super-class. Thanks to the sub-class relation, the class `PetTiger` automatically inherits all attributes of the class `Tiger`, including both methods and fields. For instances of the class `PetTiger`, it does not matter whether one of its attributes has been defined in the class `PetTiger` or in the class `Tiger`. This mechanism is called *inheritance*. Thanks to inheritance, code and information can be factorized to super-classes. The code of a super-class can be reused in all sub-classes, providing a convenient economy of representation.

In order to ensure the initialization of inherited fields, Java and FAST implement a mechanism called *constructor chaining*. A constructor always calls the constructor of the super-class. In the code example, this is done explicitly in Java by use of the reserved word `super`. The argument `n` is passed from the constructor of `PetTiger` to the constructor of `Tiger`, such that the inherited field `name` receives a value. Even if a constructor is provided neither in `Tiger` nor in `PetTiger`, Java creates the appropriate constructors and links them by constructor chaining. In FAST, the `EXTENDS`-statement itself functions as a constructor call. In the example, the argument `n` is passed to the constructor of `Tiger`, resulting in the same initialization as in Java, but with a less wordy program. The `make-instance` function in CLOS also calls the initialization functions of the super-class, but this does not

become apparent to the programmer. The initialization arguments of `make-instance` also remain valid for sub-classes, so that the function can be called for `PetTiger` and supply a value for the field `name` by `:name`.

The important property of inheritance is that an instance of a sub-class may occur wherever an instance of the super-class is expected. This property is called *subtype conformance* (Snyder, 1991). If a function requires for example a `Tiger` as an argument, then a `PetTiger` can be provided instead. Subtype conformance is particularly obvious in Java and FAST, where variables and fields have types. If a variable is of the type τ , then it may have instances of any sub-type of τ for its value. This is possible, because the content of the variable is guaranteed to have all attributes of τ – whether it is a τ or a subtype of τ . By contrast, it is in general not possible to assign a super-type to a sub-type. If a variable is for instance declared a `PetTiger`, then it may not be assigned a `Tiger`. This is not allowed, because the super-type may have fewer attributes than the sub-type, so that accesses to fields of the sub-type may become meaningless. Such an assignment results in a compiler error. In summary, the value of a variable must satisfy the *instance* relation with its type.

3.1.2. Type Refinement

A sub-concept may not only have additional attributes, but also restrict the values of existing ones. In the example, an attribute `age` will be assumed in the class `Tiger`. This attribute may have any positive integer value. By contrast, one might imagine that pet tigers do not become as old as wild tigers due to the unnatural environment. Thus, the attribute `age` needs to be restricted to a certain sub-range of the original range, say `[0,10]`. Such a restriction of a sub-class attribute is called *type refinement*. This restriction does not violate the *subconcept* relation: Every pet tiger is still a tiger, although it does not become as old as a normal tiger. Range restrictions declare those values of a field, which are admissible. Note that a sub-class may not enlarge a type of a field: If tigers had a maximum age of 20 years, then a pet tiger could not become older, because it would not be a tiger any more in the strict sense³².

Oddly enough, none of the programming languages supports value restrictions in a convenient way. At first sight, it seems as if CLOS provided a solution: In CLOS, the type of a field may be fixed by the slot-specifier `:type`. This includes not only primitive types, but also classes, conjunctions of types and even types satisfying a given condition. Thus, the constraint of having an integer value between 0 and 10 can simply be expressed by the type `(integer 0 10)`. However, this facility does not solve the problem: First, a sub-class declaration may enlarge the type of a field, which is ontologically implausible³³. Second, the type declaration does not have any effect on the program. If the field receives a value, which does not fit the type, the behaviour is just "undefined" (Steele, 1990).

Apart from some special cases, the typing in Java and FAST is checked by the compiler. This entails that sub-classes cannot restrict the type of a field: If a variable is of type `Tiger` and the value `15` is assigned to its `age`, then this should in general be allowed. Nevertheless, if the variable has a `PetTiger` for its value, then this should result in an error, because the range of `age` for `PetTiger` is 0 to 10. As the compiler cannot know whether the variable contains a `Tiger` or a `PetTiger` at run-time, it cannot check the type match. As a result, fields of sub-classes must always have the same type as the corresponding fields of the super-class in Java and FAST. Furthermore, neither Java nor FAST allows range typing like CLOS.

Java and FAST can use constructor chaining for the verification of field values. Each class has a constructor, which usually assigns initial values to fields. The constructor may thus be used to check at run-time whether the initial values fulfil certain conditions. The constructor of `PetTiger` may for instance verify if the `age` is contained in `[0, 10]`. A constructor automatically calls the constructor of the super-class. If both the constructor of the sub-class and the constructor of the super-class check certain conditions for initial values, then this results in a conjunction of these conditions. Thus, constructor chaining provides a convenient way of checking initial values at run-time in an ontologically plausible way. Nevertheless, only accessor methods can prevent the assignment of invalid values after the creation of the object.

³² This constraint does not prevent sub-classes from having other field values than their super-classes. It just states that ranges of super-concepts must include ranges of sub-concepts.

³³ Although the type of a field is afterwards computed as an intersection of types, this does not prohibit an enlarging declaration.

Although type constraints can be checked at run-time like in Java and FAST or can be declared like in CLOS, none of the languages is able to draw conclusions from these constraints. If the conditions restrict the set of possible values to a singleton, none of the languages actually assigns this value to the field. Schneider-Patel criticises that it is impossible to deduce attribute values from type constraints in common object-oriented programming languages (1991). This is due to a procedural view of knowledge rather than a declarative one (Bench-Capon, 1990). In summary, none of the languages is designed for an ontologically correct and verified type refinement.

3.1.3. Overwriting Attributes

Up to now, sub-classes only narrowed the set of possible values for attributes. Sometimes, it is useful to initialize fields of sub-classes differently from fields of super-classes. Initial values for fields are called *default values*³⁴. They provide an assumption about a field value, which has not yet been filled explicitly. In the example, pet tigers shall have a default age of zero, as they are often bought when they are babies. By contrast, normal tigers shall have a default age of ten. CLOS, Java and FAST provide means for giving initial values to fields. It is assumed that the initial value of `age` in the class `Tiger` is set to 10. Now, the same declaration of `age` is repeated in the class `PetTiger` to assign the initial value 0:

CLOS	Java	FAST
<code>(age :initform 0)</code>	<code>int age=0;</code>	<code>VAR age 0</code>

CLOS and FAST behave the expected way: The field `age` receives the initial value 0. If a `PetTiger` is created, it is by default a baby. This procedure is called *overwriting a field* and establishes a difference between instances of the super-class and instances of the sub-class. Thanks to subtype conformance, it is possible to store instances of the sub-class in a super-class variable. Despite of that, the sub-class instance keeps its class-specific attributes and values. Thus, instances of the sub-class may be handled as instances of the super-class, but still behave according to their true class membership. Making use of this property is called *abstraction*. In the example, the difference just consists in the assumption of different default values and vanishes as soon as new values are assigned to the field.

Java treats the re-declaration of fields differently: It introduces a new field `age` for `PetTiger`, which exists along with the inherited one from `Tiger`. If `kitty` is an instance of `PetTiger`, then `kitty` has two fields named `age`. If `kitty` is accessed via a variable of type `PetTiger`, then the field `age` has the expected value of 0. If `kitty` is accessed via a variable of type `Tiger`, the value of `age` is 10. The following lines illustrate the phenomenon:

```
// Create pet tiger Kitty
PetTiger kitty=new PetTiger("Kitty");
// Print Kitty's age
System.out.println(kitty.age); // Prints 0
// Declare a variable of type Tiger, assign Kitty to it
Tiger kitty2=kitty;
// Print Kitty's age again
System.out.println(kitty2.age); // Prints 10
```

This mechanism is called *shadowing* in Java. The correct way of changing the default value for `age` would be to put a value assignment into the constructor of `PetTiger` instead of re-declaring the field. In programs, the mechanism of shadowing is rarely used, as it causes the advantages of inheritance and abstraction to vanish. In summary, CLOS, Java and FAST allow the change of default values for sub-classes, although some care is required in Java.

³⁴ In terms of (Padgham, 1991), this could correspond to the "usual characteristics" of a type

3.1.4. Overwriting Shared Attributes

Shared attributes may also have different values for sub-classes. In the example, the default value for the dangerousness of tigers is 100%, whereas it will be reduced to 10% for pet tigers. To do so, the attribute is re-declared in the class `PetTiger`:

CLOS	Java	FAST
<pre>(dangerous :allocation :class :initform 10)</pre>	<pre>static int dangerous=10;</pre>	<pre>SHARED dangerous 10</pre>

Again, CLOS and FAST behave the expected way: An instance of the class `PetTiger` has a value of 10 for the field `dangerous`. Overwriting the shared field causes the field to exist twice, once for each class. For direct instances of `Tiger`, the attribute refers to the original field of the class `Tiger`. For direct instances of `PetTiger`, it refers to the overwritten field declared in the class `PetTiger`. This also applies if an instance of `PetTiger` is treated as an instance of `Tiger`. Thus, CLOS and FAST allow abstraction also for shared fields. By contrast, Java shadows shared fields. The field `dangerous` also exists once for each class, but an instance of `PetTiger` can refer to both of them. Depending on the type of the holding variable, the value of `dangerous` for the very same instance is either 10 or 100. Abstraction with shared fields is not possible in Java. Unlike for non-shared fields, there is no way to inherit a shared field without a re-declaration and change its value for the sub-class. If `PetTiger.dangerous` were changed, `Tiger.dangerous` would be changed as well.

A solution would be to declare the field in both classes. Then, a method `getDangerous` has to be written, which returns `Tiger.dangerous` for tigers and `PetTiger.dangerous` for pet tigers. As this solution is quite complicated, it can be said that the design of Java does not support true inheritance of shared attributes. It becomes obvious that static fields in Java merely fulfil the purpose of global variables: They are accessible from all methods and functions within the class and they can be accessed without the presence of an instance. They are inherited to a sub-class, but accessing the field via the sub-class is just a syntactic variant. On the other hand, if the field is re-declared in the sub-class, then this results in two different fields. As abstraction is not supported, the field a static variable refers to can always be determined at compile-time. In summary, static variables lack important properties of shared fields and should thus rather be considered global variables within their scope. Java lacks powerfulness of design for shared attributes.

3.1.5 Overwriting Methods

Up to now, normal attributes and shared attributes have changed their values for the sub-concept. In addition to that, action attributes may also be different for sub-concepts. This phenomenon is called *polymorphy*. In the example, a pet tiger, which is told to roar, will purr. CLOS, Java and FAST all support polymorphy by re-declaration of the method for the sub-class. CLOS defines the new `roar` method outside the class, whereas Java and FAST declare it inside the class:

CLOS	Java
<pre>(defmethod roar ((a PetTiger)) (print "mmmr")))</pre>	<pre>void roar() { System.out.println("mmmr"); }</pre>
FAST	
<pre>FUNCTION roar (PRINT "mmmr")</pre>	

If `kitty` is a `PetTiger` and the `roar` method is called for `kitty`, then the string "mmmr" will appear on the display. Concerning polymorphy, CLOS, Java and FAST behave the same way: The

method is *overwritten*³⁵. All instances of the class `PetTiger` will use the re-declared method, independent of the type of the holding variable. The call to an overwritten method necessitates a so-called *dynamic method look-up* by the run-time system. It is astonishing that this is the case in Java, too: Re-declared fields exist twice for an instance, whereas re-declared methods exist once. This is an incoherence, which can be seen as a drawback in the domain of expressiveness.

3.1.6. Inference Mechanisms

In Ontology, the focus is mostly on shared attributes. For the time being, only concept graphs in form of trees will be investigated, i.e. every concept has exactly one super-concept. By default, it is assumed that a sub-concept has the same value for an inherited shared attribute as its super-concept. Only if specified otherwise, shared fields of sub-concepts take another value than the inherited one. This assumption allows statements about the values of shared attributes, although these values have not been defined explicitly. This technique is called *default reasoning*. In the example of tigers, it would allow to assume that circus tigers are as dangerous as tigers, because the dangerousness of circus tigers has not yet been specified otherwise³⁶. The problem of determining the value of a shared field for a given instance is called the *inference problem*. As Bibel, Hölldobler and Schaub remark, the main purpose of ontological representations is serving these inferences (1993).

CLOS, Java and FAST allow solving the inference problem. In terms of programming languages, it translates to an evaluation of the shared field for a given object. Given a `PetTiger kitty`, the inference problem for `dangerous` can be programmed simply by

CLOS	Java
<code>(print (slot-value kitty 'dangerous))</code>	<code>System.out.println(kitty.dangerous);</code>
FAST	
<code>PRINTINT (dangerous kitty)</code>	

CLOS is the only language, which determines the value of a shared field by a dynamic search in the concept tree. If `slot-value` is called with an instance, CLOS checks whether the field has been defined in the class of the instance. If this is not the case, CLOS checks the super-class and again the super-classes, until the field has been found. If the tree has a height of n , then the operation is an element of $O(n)$. Although this time can be neglected with modern computers and with moderate concept trees, Sowa remarks that execution time may become relevant with larger databases or more inferences (2000).

Java, by contrast, determines the field value in constant time. To be precise, it could even know the value at compile-time, if the field was declared unchangeable. This is due to the nature of static variables, which resembles the one of global variables. The advantage of quick access is questioned by the lacking abstraction facility. If the instance is not given by a variable of the same class, then the inference fails. If `kitty` is stored in a variable of type `Tiger`, then `kitty` will become dangerous. This property of Java disrupts the inference drastically.

In FAST, attribute fields are also determined in constant time. Unlike Java, FAST allows abstraction with shared fields, so that the instance will always keep its value, independent of the variable, which refers to it. In summary, only CLOS and FAST allow a real solution of the inference problem. Furthermore, FAST can solve the problem in constant time.

Unlike a classical ontology, a programming language allows changing the value of a shared attribute. As Snyder points out, this allows the dimension of time to be taken into account (1991). In general, the logic often used in Ontology is static, whereas programming languages can express a change of values in the course of the program. However, Bench-Capon criticizes that the semantics of overwriting attributes is not as well defined as the ontological concept of default reasoning (1990). Furthermore, he remarks that an attribute of a concept does not allow any implications, because it

³⁵ America demands that methods should be specialized rather than overwritten (1991).

³⁶ Non-monotone logics deal with the theoretical foundations of these situations (cf. e.g. Marek and Truszczynski, 1993).

might always be overwritten in a sub-concept. In general, the variability of values even prohibits implications in a strict sense.

3.1.7 Chapter Summary

In this chapter, various differences of pet tigers and tigers have been established. Attributes have been added and attribute values have been overwritten, but the `subconcept` relation between the concepts `PetTiger` and `Tiger` has not been violated. If an attribute is not overwritten, then it is simply inherited from its super-concept. This applies to the default-value of a normal field, to the value of a shared field and to the implementation of a method. If the attribute is re-declared in the sub-class, then all instances of the sub-class always refer to the re-declared attribute. This applies to methods, fields and shared fields in CLOS and FAST. In Java, this only applies to methods, since a re-declaration of fields results in the creation of two distinct fields, which are both accessible from one object. This is a violation of the uniformity of methods and fields and causes Java to appear less expressive than CLOS and FAST in the issue.

The combination of inheritance and overwriting allows solving the inheritance problem by the use of programming languages. It has become apparent that the shadowing mechanism of Java prevents a use of static fields as shared fields. The design of Java does not support shared fields in the sense of Ontology. Static fields are just scoped global variables. As a result, Java is not an appropriate language for the inference problem. CLOS and FAST both allow the solution of the inference problem. CLOS needs linear time to do so, whereas FAST only requires constant time.

3.2. Multiple Inheritance

In this chapter, the concept `CircusTiger` will be introduced. It is a sub-concept of `Tiger` and a sister-concept of `PetTiger`. The concept of circus tigers will allow creating an interesting scenario: A common sub-concept of `CircusTiger` and `PetTiger` can be defined. For the first time, a concept will have two super-concepts. This chapter will show how the programming languages deal with this situation. As Java follows another approach than CLOS and FAST, the language will be treated apart.

3.2.1. A Sister Concept in CLOS and FAST

In CLOS and FAST, the class `CircusTiger` can be defined similarly to the class `PetTiger`. It extends the class `Tiger`. The primary quality of circus tigers is that they are able to perform stunts. This is why the class `CircusTiger` will have a method `stunt`. Since circus tigers are owned by a circus, they also have an attribute `owner`. The method `roar` is not modified; it is inherited from `Tiger`. By contrast, the attribute `dangerous` will be changed: Circus tigers are dangerous to a degree of 50%. In CLOS and FAST, the declaration of the new class looks as follows:

CLOS	FAST
<pre>(defclass CircusTiger (Tiger) ((owner) (dangerous :initform 50))) (defmethod stunt ((a CircusTiger)) (print "Performing stunt")))</pre>	<pre>CLASS CircusTiger n String (EXTENDS Tiger n VAR owner "" SHARED dangerous 50 FUNCTION stunt (PRINT "Performing stunt")))</pre>

The class `PetTiger` and the class `CircusTiger` share the same attribute name `owner`, although its intention is different: In `PetTiger`, it refers to the animal lover, whereas in `CircusTiger`, it refers to the circus. Up to now, these declarations are not conflicting: As no `PetTiger` is a `CircusTiger` and no `CircusTiger` is a `PetTiger`, instances of both classes are well distinguished. In the typed language FAST, a warning is displayed if a variable declared `PetTiger` holds a `CircusTiger` or vice versa. Thus, no ambiguity concerning `owner` is possible. If a `PetTiger` or a `CircusTiger` is assigned to a variable of type `Tiger`, then the conflicting attribute `owner` is not visible any more. In summary, the two attributes with the same name are well defined within their scope.

3.2.2. A Common Child Concept in CLOS and FAST

Now, the concept of pet circus tigers will be defined. Pet circus tigers are tame, stunt performing tigers; they are both circus tigers and pet tigers. In the ontology, this means that the concept `PetCircusTiger` has two super-concepts, namely `CircusTiger` and `PetTiger`. This phenomenon is called *multiple inheritance* – as opposed to the *single inheritance*, which has been investigated so far. In the example, the concept `Tiger` occurs twice as a super-concept of `PetCircusTiger`: First as a super-concept of `PetTiger` and then as a super-concept of `CircusTiger`. If there are two paths in the concept graph from a sub-concept to a super-concept, then the super-concept is called a *common ancestor concept*. CLOS and FAST allow multiple inheritance with classes. Thus, the concept of pet circus tigers can be programmed as follows:

CLOS	FAST
<pre>(defclass PetCircusTiger (PetTiger CircusTiger) ())</pre>	<pre>CLASS PetCircusTiger n String (EXTENDS PetTiger n EXTENDS CircusTiger n)</pre>

In CLOS, the list of extended classes simply contains the two super-classes. No further modifications or declarations are needed. In FAST, the two **EXTENDS**-clauses declare the two super-classes. Each of them has to be a fully qualified constructor call and thus, the argument *n* has always to be provided. Furthermore, FAST requires common ancestor classes to be identified explicitly. Any class, which shall be able to have sub-classes with a common child class, has to be declared *virtual*. This is done by exchanging the reserved word **CLASS** by the reserved word **VCLASS**. In the example, the class **Tiger** has to be declared virtual. This is a drawback concerning the expressiveness, as the code of an existing class has to be changed if a new class is added.

In summary, the creation of a class with two ancestors is supported by the design of CLOS and FAST. Up to now, no error results due to the conflicting attribute **owner** of the two parent classes.

3.2.3. Interfaces in Java

Java does not allow multiple inheritance with classes. If the class **CircusTiger** were declared as in CLOS and FAST, then Java would not permit a class **PetCircusTiger**, which inherits from both of them. Java offers another language component to express multiple inheritance, called *interfaces*. Both interfaces and classes represent concepts in the program source code³⁷. Unlike classes, interfaces are only concept specifications without an implementation. They look like classes, but may only contain constant fields and method signatures³⁸. They may not contain any method body. Thus, interfaces only specify a certain behavior, but do not provide an implementation of this behavior. Interfaces may serve as types. They may extend other interfaces as classes extend other classes. In particular, they may also extend multiple interfaces, resulting in a kind of multiple inheritance for interfaces. Unlike classes, interfaces cannot be instantiated: It is impossible to create a direct instance of an interface, because interfaces only serve as abstract specifications.

If a class provides all methods specified by an interface, then the class is said to *implement* this interface. This relation is expressed by the keyword **implements** in the declaration of the class. In the ontology, this corresponds to a **subconcept** relation between the class and the interface³⁹. Every class (except for **object**) has exactly one direct super-class. Nevertheless, every class may implement multiple interfaces. By contrast, an interface may not extend a class. These constraints can be expressed formally: Let **class** be the property of a concept of being programmed as a class. Let **interface** be the property of a concept of being programmed as an interface. Then the following holds in the ontology:

$$\begin{aligned}
 \forall x: & \quad \text{class}(x) \Rightarrow \text{concept}(x) \\
 \forall x: & \quad \text{interface}(x) \Rightarrow \text{concept}(x) \\
 \neg \exists x, y: & \quad \text{direct-instance}(x, y) \wedge \text{interface}(y) \\
 \forall x: & \quad \text{class}(x) \wedge x \neq \text{object} \Rightarrow \\
 & \quad \exists ! y: \text{direct-superconcept}(y, x) \wedge \text{class}(y) \\
 \forall x: & \quad \text{interface}(x) \Rightarrow \neg \exists y: \text{subconcept}(x, y) \wedge \text{class}(y)
 \end{aligned}$$

In the view of Java, interfaces represent additional qualities of classes. They are designed to mark out certain classes⁴⁰. If a class implements an interface, then this means that the class is guaranteed to provide the fields and methods specified in the interface. It will now be shown how interfaces can solve the problem of pet circus tigers.

³⁷ In the view of Java, interfaces represent properties, but according to the terminology of this paper, they correspond to concepts.

³⁸ They may also contain inner classes and interfaces, but these concepts are not dealt with in this paper.

³⁹ Since interfaces are intended to bear names of adjectives, the Java terminology allows talking of a class as *being an X*, if X is an interface. This alludes to the *is-a* relation.

⁴⁰ This corresponds to the idea of *mix-in classes* (cf. e.g. Bracha and Cook, 1990).

3.2.4. A Sister Concept in Java

In order to represent pet circus tigers, one of its parent concepts `PetTiger` and `CircusTiger` has to be interpreted as an "additional quality" of tigers. Here, the concept `CircusTiger` is arbitrarily chosen. The ability to perform a stunt is seen as a supplemental property for tigers. Now, the concept can be programmed as an interface. The interface will be called `StuntPerforming`. As soon as the interface has been declared, the class for pet circus tigers can be created: It will extend `PetTiger` and implement `StuntPerforming`. Thus, it realizes two **subconcept** relations. This is the only way to allow multiple inheritance with a class in Java⁴¹.

It will be tried to design the interface `StuntPerforming` as close as possible to the class `CircusTiger`. Interfaces are less powerful than classes. The interface `StuntPerforming` may not provide an implementation for the method `stunt`. Hence, it will only specify the method. Furthermore, the interface cannot declare the field `owner`, as this is not necessarily a constant field. Usually, non-constant fields are specified by their accessor methods: If a class implements the reader method and the writer method for a field, and if this implementation follows the semantic conventions of accessor methods, then this is functionally equivalent to a class with the field itself. Hence, the interface `StuntPerforming` will specify the methods `getOwner` and `setOwner`. The attribute `dangerous` can be taken over, if it is accepted to be constant. Overall, the new interface looks as follows:

```
interface StuntPerforming {
    static int dangerous=50;
    void stunt();
    String getOwner();
    void setOwner(String o);
}
```

In order to allow multiple inheritance in Java, it is sometimes necessary to program concepts as interfaces instead of classes. Interfaces entail a number of inconveniences for the representation of ontological structures. First, it has to be decided which concepts are "full concepts" and which concepts are "additional qualities". From an ontological point of view, this distinction is not always canonical. In the example, one could have opted to interpret *being a pet* as an additional quality as well. Second, an interface cannot be a sub-concept of a class. Thus, it is for example impossible to express that any `StuntPerforming` entity be a `Tiger`. Third, interfaces may not provide method bodies. If multiple animal classes implement the interface `StuntPerforming`, then the code for `stunt` has always to be rewritten, although it might actually be the same. By contrast, the fields of an interface are truly passed to the implementing class. Fourth, interfaces only allow constant attributes with an initial value. Without accessor methods, it is impossible to require an implementing class to provide a non-constant field. Default values for non-constant fields cannot be represented at all. The lack of method bodies prohibits the type refinement of field values by run-time checks. Fifth, interfaces cannot be instantiated. In order to allow circus tigers to be created, a class `CircusTiger` would have to be written, which implements `StuntPerforming`. This class could then be used for the creation of circus tigers. However, the additional class causes an unnecessary multitude of equivalent concepts. In summary, interfaces are less useful than classes for the representation of ontological structures⁴².

3.2.5. A Common Child Concept in Java

Now, the concept `PetCircusTiger` can be implemented in Java. It will be represented by a class called `PetCircusTiger`, which extends `PetTiger` and implements `StuntPerforming`. Due to the interface specification, it needs to define the methods `getOwner`, `setOwner` and `stunt`. Although visibility modifiers have been omitted so far, they need to be specified for these methods. This is because an interface implicitly demands `public` visibility for all specified methods.

⁴¹ Strictly speaking, the notion *multiple inheritance* is not appropriate for Java, but in this paper, it only denotes the phenomenon of multiple direct super-concepts.

⁴² Interfaces have been introduced in Java to avoid the problems of multiple inheritance. See (Gosling et al., 2000) for a description of the problems of multiple inheritance, which appear with interfaces.

```

class PetCircusTiger extends PetTiger implements StuntPerforming {

    public void stunt() {
        System.out.println("Performing stunt");
    }

    public String getOwner() {
        return owner;
    }

    public void setOwner(String o) {
        owner=o;
    }

    PetCircusTiger(String n) {
        super(n);
    }

}

```

The interface `StuntPerforming` demands the presence of the methods `getOwner` and `setOwner`. The class already inherits the attribute `owner` from `PetTiger`. In order to ensure the conventional equivalence of an attribute and its accessor methods, it has been chosen to have the accessor methods `getOwner` and `setOwner` refer to the field `owner`. The next chapter will point out the consequences and alternatives of this decision.

The class `PetCircusTiger` truly implements two `subconcept` relations: Due to subtype conformance, any variable of type `PetTiger` may hold an instance of `PetCircusTiger`. Subtype conformance exists for interfaces as well: Any variable of type `StuntPerforming` may hold an instance of `PetCircusTiger`. Although the concept of stunt performing tigers was intended to be a sub-concept of tiger, the interface `StuntPerforming` cannot be a sub-concept of `Tiger`. Thus, instances of a class implementing `StuntPerforming` may not be simply assigned to a variable of type `Tiger`. Such an assignment requires an explicit trans-typing. This shows that interfaces are not intended for multiple inheritance with a common ancestor class.

3.2.6. Chapter Summary

In this chapter, the problem of multiple inheritance was investigated. Whereas CLOS and FAST support multiple inheritance of classes, Java requires the use of interfaces. The concept of interfaces necessitates a strict distinction of concepts and additional qualities. Due to the similarity of qualities and concepts, such a distinction is not always ontologically plausible. Furthermore, interfaces do not provide the power of classes concerning inheritance and attribute declarations. In summary, it has been pointed out that interfaces are less useful than classes for Ontology. As Java provides no other solution for multiple inheritance than interfaces, it lacks powerfulness of design for this issue.

3.3. Problems of Multiple Inheritance

This chapter will present a number of ontological problems, which arise due to the introduction of the concept `PetCircusTiger`. First, the attribute `owner` is ambiguous for pet circus tigers, because `owner` is defined in the super-concept `PetTiger` and in the super-concept `CircusTiger`. It will be shown how the programming languages handle this conflict. Furthermore, the attribute `name` is inherited twice by the concept `PetCircusTiger`, because both super-concepts inherited it independently from the common ancestor concept `Tiger`. It will be investigated whether this fact causes the programming languages to implement the attribute twice. Last, the question will be to what degree pet circus tigers are dangerous, as they inherit from two types of tigers with two degrees of dangerousness. It will be explored, for which value the programming languages decide. This treatise will lead to a general discussion of the inference problem in the object-oriented programming languages.

3.3.1. Ambiguous Inherited Attributes

Both the concept `PetTiger` and the concept `CircusTiger` have an attribute `owner`. For pet tigers, the attribute refers to the proprietor of the pet, whereas for circus tigers, it refers to the circus. The concept `PetCircusTiger` inherits from both `PetTiger` and `CircusTiger`. This causes an ambiguity of the attribute `owner` for the concept `PetCircusTiger`. If two super-concepts both have an attribute of the same name, which is not contained in a common super-class, then this attribute is called an *ambiguous inherited attribute* (Gosling et al., 2000). Two situations can be imagined: The two attributes of the two super-concepts can be intended for different purposes. Then the attributes are distinct. This also applies if an instance of the child concept has two different values for the attribute, depending on the role it takes. In the example, the owner of a pet circus tiger in its role as a circus tiger is the circus. The owner of the tiger in its role as a pet may be a clown. For convenience, attributes of this kind will be said to have *different meanings*. A proper solution for this conflict would be to assign unique names to the attributes. Another situation is given if the two attributes are intended for the same purpose. If a child concept inherits from two concepts, which have two attributes with the same value, then the child concept should implement this attribute only once. In this case, the attributes will be said to have the *same meaning*. A proper solution for this situation would be to factorize the common attribute to a new concept. The two sister concepts can inherit the common attribute from this new concept and thus indicate that the meaning is the same. If the attribute is neither factorized nor renamed, the conflict appears. CLOS, Java and FAST handle it in completely different ways.

In CLOS, one ancestor of an ambiguous inherited attribute hides the other. If an ambiguous attribute is accessed, then the attribute of the class, which is mentioned first in the super-class list, is taken. In the example, the class `PetTiger` precedes the class `CircusTiger` in the super-class list of `PetCircusTiger`. Hence, a use of the attribute `owner` with an instance of `PetCircusTiger` will always refer to the field defined in `PetTiger`. If the class `PetTiger` provided the default value "A" for the field `owner` and the class `CircusTiger` provided the value "B", then the field of a new instance of `PetCircusTiger` would have the value "A". In CLOS, the super-class mentioned first predominates over the other super-classes. Whenever a conflict arises, the contribution of the first super-class is weighted more. A vague ontological interpretation would be that a pet circus tiger is a pet tiger to a higher degree than a circus tiger. The programmer may choose which class shall be preferred to resolve ambiguous inheritance conflicts. In summary, an ambiguous inherited attribute exists only once in CLOS. The language is thus intended for cases, where two attributes of the same name have the same meaning. However, CLOS does not provide a solution for cases where this assumption is false (like in the example of `owner`). There is no way to access both inherited attributes, if they bear the same name.

Java does not allow multiple inheritance with classes. Consequently, the conflict of ambiguous inherited attributes may only appear in combination with an interface. Interfaces only allow methods and constant fields. With methods, the conflict does not arise at all: The purpose of a method specification in an interface is not to pass the method to the class, but rather to demand that the class provide it. The compiler is also satisfied if the class inherits the required method. Thus, it is assumed that the inherited method fulfils the same function as the method specified by the interface. This

assumption may fail: In the example, the method `getOwner` requested by the interface `StuntPerforming` is intended to return the circus, whereas a method `getOwner` eventually defined in `PetTiger` returns the proprietor of the pet. Since such a situation contradicts the intention of interfaces, Java unifies the methods.

With ambiguous inherited constant fields, the situation is different. Unlike methods, constant fields are truly passed to the implementing class by an interface. The conflict of an ambiguous inherited attribute may thus arise, if both an implemented interface and a super-class provide the same constant field. In the example, the attribute `dangerous` is such an ambiguous inherited constant field. It is inherited from the class `PetTiger` and from the interface `StuntPerforming`. The problem is solved very elegantly in Java: The two fields can be accessed separately by trans-typing the instance either to the interface or to the class. Assuming a pet circus tiger `shirkitty`, accesses to the different fields would look as follows:

```
((PetTiger)shirkitty).dangerous
((StuntPerforming)shirkitty).dangerous
```

If this type specification is omitted, the compiler announces an error. Thus, Java avoids the conflict by clearly distinguishing the two attributes. If the two attributes have the same meaning and the programmer wishes to have one single field, then she or he may re-declare the attribute in the class `PetCircusTiger`. Thus, the programmer has the choice of explicit disambiguation or re-declaration. Although this conflict resolution seems very powerful, it has to be kept in mind that the re-declaration of the field loses its effect as soon as the instance is stored in a variable of a super-type. Consequently, it cannot be considered a proper solution. Furthermore, the technique is restricted to constant fields.

In FAST, the approach of disambiguation in Java is generalized to all ambiguous inherited attributes. A class implements an ambiguous inherited attribute twice. The different fields can be accessed by trans-typing the instance. In the example of `shirkitty`, this also applies to the attribute `owner`:

```
(owner (shirkitty : PetTiger))
(owner (shirkitty : CircusTiger))
```

The colon re-interprets an instance as an instance of a given class. If this re-interpretation is not given, the compiler announces an error. This shows that in FAST, both attributes with their respective intention are preserved. FAST does not provide a facility for unifying ambiguous inherited attributes, if they have the same meaning. Thus, the language is intended for cases, where two attributes with the same name in two different classes always have two different meanings.

As it has been shown, all languages are guided by different assumptions about ambiguously inherited attributes. In CLOS, it is assumed that they have the same meaning, whereas FAST handles them as if they had different meanings. In Java, methods are assumed to have the same meaning, whereas constant fields are distinct. It cannot be determined, which assumption is more likely to be correct. In general, distinct attributes should have different names and equal attributes should be factorized.

3.3.2. Multiply Inherited Attributes

The concept `Tiger` provides the attribute `name`. It is inherited by `PetTiger` and `CircusTiger`. As a result, the concept `PetCircusTiger` inherits the attribute in two ways. An attribute inherited from a common ancestor concept is called a *multiply inherited attribute* (Gosling et al., 2000). In most cases, the child concept is desired to possess the multiply inherited attribute only once. In the example, a pet circus tiger is intended to have one single name. Nevertheless, it can be imagined that both attributes shall be maintained. In the example, the tiger could have a nickname in its role as a pet tiger and an artist name in its role as a circus tiger. Some programming languages share this point of view, but in CLOS, Java and FAST, the attribute `name` exists only once.

In CLOS, a multiply inherited attribute is unique. The attribute identifier always refers to the field declared in the common ancestor class. Thus, a use of the field `name` with an instance of the class `PetCircusTiger` will refer to the field inherited from the class `Tiger`. In Java, multiply inherited

attributes do not exist in classes, because the language does not support multiple inheritance with classes. A class attribute is passed down a unique way in the class hierarchy. In the example, the attribute `name` is passed down from `Tiger` to `PetTiger` to `PetCircusTiger`. Any use of the field is unambiguous. FAST does support multiple inheritance, but also creates the field just once. An instance of `PetCircusTiger` has a single attribute `name`, which maintains its uniqueness even if the instance is stored in a variable of another type. Unlike CLOS, FAST does not perform a dynamic search at run-time to access instance attributes. With multiply inherited fields, this is only possible if FAST knows in advance whether a class takes the role of a common ancestor concept. This explains the need for the virtual declaration of the class `Tiger`. In summary, CLOS and FAST share the view that multiply inherited attributes exist only once, whereas Java does not know multiply inherited class attributes at all.

3.3.3. Inference Mechanisms with Multiple Inheritance

The multiple inheritance of the concept `PetCircusTiger` provides an interesting setting for the inference problem. The attribute `dangerous` is defined in the concept `Tiger`, it is inherited by both `PetTiger` and `CircusTiger` and it is multiply inherited by `PetCircusTiger`. The default value is different for each of the concepts involved: Tigers are dangerous to a degree of 100%, pet tigers to a degree of 10% and circus tigers to a degree of 50%. Now, the question arises to what degree pet circus tigers are dangerous. The problem of the value of a multiply inherited attribute is fundamental in Ontology⁴³ and it will be shown how the programming languages solve it. It has already been pointed out that Java is less adequately designed for the inference problem. Furthermore, Java does not support true multiple inheritance. Consequently, the solution of the inference problem will only be programmed in CLOS and FAST.

As in the case of single inheritance, the determination of the value of an attribute translates into an evaluation of a shared field in the programming languages. If `shirkitty` is an instance of the class `PetCircusTiger`, then the following program evaluates the attribute `dangerous`:

CLOS	FAST
<code>(print (slot-value shirkitty 'dangerous))</code>	<code>PRINTINT (dangerous shirkitty)</code>

Both programs display a single value: CLOS displays 10, whereas FAST displays 50. Thus, CLOS took over the value from the class `PetTiger`, whereas FAST took over the value from `CircusTiger`. None of the languages decided for the value of the class `Tiger`. This value is never considered, because it is overwritten in both sub-concepts. Furthermore, none of the languages calculated the mean value of the values provided by the direct super-concepts. Although this would be a plausible solution in the ontology, the approach could not be generalized to fields of other types. Moreover, neither CLOS nor FAST required an explicit disambiguation or displayed two values, because both languages consider a multiply inherited attribute unique. In the light of these constraints, the only solution for the inference problem is to decide for one of the ancestor classes to provide the field value. In order to determine this class, all ancestor classes are taken into consideration. The first class, which defines a value for the attribute, is chosen. The sequence, in which the classes are traversed, is sophisticated. In order to explain how CLOS and FAST find the value of the attribute, the following more complex concept graph will be used:

⁴³ It corresponds to the famous *Nixon-Diamond*. A detailed description of this structure can be found in (Stein, 1991a).

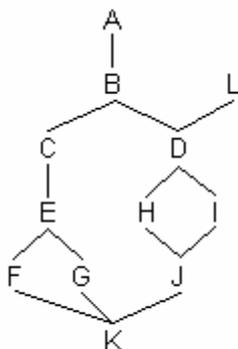


Figure 2: A more complex concept graph

The task is to determine the default value for an attribute of an object. The class, which the object is a direct instance of, will be called the *direct class*. The restriction of the class graph to the direct class and all of its super-classes is called the *ancestor graph*. Among the ancestor classes, some may define the attribute in question, and some may not. Some may provide a default value for it, whereas others may not. The above illustration shows the ancestor graph of the direct class κ . The left-to-right arrangement of super-classes in the picture is intended to reflect the order, in which the classes are extended in the program. This means that the class \jmath , for example, first extends the class \mathfrak{H} and then the class \mathfrak{I} in the program code.

In CLOS, any access to a field causes a search for the field at run-time. Starting from the direct class, the ancestor graph is traversed until the field has been found. Three rules govern this process:

1. A sub-class has priority over its super-classes
As it has already been discussed, this principle is an ontological necessity: A sub-concept is more specific than a super-concept, so that the value defined in the sub-concept should be preferred.
2. A super-class mentioned earlier in the super-class list has priority over a class mentioned later
As discussed in the previous section, this principle establishes a preference if two direct super-concepts define conflicting attributes or values.
3. Families shall be visited together

A *family* is a lattice in the ancestor graph, which has links entering it only in the lower bound and links exiting it only in the upper bound. In the example, $\{\jmath, \mathfrak{H}, \mathfrak{I}, \mathfrak{D}\}$ would be a family. As the **standard-class** constitutes the upper bound of any class graph in CLOS, the ancestor graph is itself a family, which consists of multiple sub-families.

The list of classes visited by a complete traversal is called the class precedence list. CLOS uses the following algorithm: Starting from the direct class, it performs a depth-first left-to-right search in the ancestor graph. Whenever it is about to encounter a common ancestor class, which it will encounter again, it performs backtracking. This corresponds to a normal depth-first search in the ancestor graph, if all links between a class and its sub-classes are cut except for the right-most one. In the example graph, the links $(\mathfrak{B}, \mathfrak{C})$, $(\mathfrak{E}, \mathfrak{F})$ and $(\mathfrak{D}, \mathfrak{H})$ would be cut. Then, a standard depth-first left-to-right search can be performed. In the example, the traversal would be **KFGECJHIDBAL**. This class precedence list can also be computed as follows: One takes the list resulting from a complete depth-first search (**KFECBAGECBAJHDBALIDBAL**) and eliminates the first appearances of repeated common ancestor classes. The following formulas describe this process. The usual list functions **first**, **but-first**, **reverse**, **list**, **append** and the predicate **member** are taken for granted. **nil** denotes the empty list and the function **sc1** returns the super-class list of a given class. As in C and Java, the question mark ? returns a conditional value:

```

class-precedence-list(c) := elim-rep(df-search(c))

df-search(c) := append(list(c), df-searchh(sc1(c)))

df-searchh(s) := (s=nil) ? nil :
                append(df-search(first(s)), df-searchh(but-first(s)))
  
```

```

elim-rep(l) := (l=nil) ? nil :
             member(first(l),but-first(l)) ?
             elim-rep(but-first(l)) :
             append(list(first(l)), elim-rep(but-first(l)))

```

The first class in this class precedence list, which defines the attribute in question, is taken. It can be seen that this algorithm satisfies the above principles: The first two principles are implemented in the functions `df-search` and `df_searchn` by the preference for the sub-class and for the first super-class. Furthermore, the depth-first approach in the graph with cut links always visits families together, because the upper bound of a family is only reached after having visited all members.

In FAST, attribute accesses are done in constant time. No search in the class graph is necessary. The default value for an attribute is set by the constructors. Thus, the order, in which the constructors are called during the creation of the object, determines the default value of the field. The constructor, which last overwrites the value, establishes it. A constructor first calls the constructors of its direct super-classes in the order of the `EXTENDS`-statements. Each super-class calls again the constructors of its super-classes, so that all constructors of all classes in the ancestor graph are called. During this process, no constructor is executed a second time. The constructors always call their super-constructors first and then run their own body. In the example, the constructor bodies would be executed in the order `ABCEFGLDHIJK`. As a latter constructor destroys the value stored by a prior constructor, the class precedence list in the sense of CLOS would be the reverse of this list. The class precedence list can be computed as follows: One starts with the direct class and adds the class precedence lists of the direct super-classes. Since a later super-class overwrites the value of a prior super-class, the super-classes have to be looked through in reverse order. Last, repeated classes have to be eliminated. If a class appears twice in the list, then the first occurrence in the list results from a second call of its constructor. As a constructor is never called a second time, this first occurrence has to be deleted.

```

class-precedence-list(c) := elim-rep(full-list(c))

full-list(c) := append(list(c),full-lists(reverse(scl(c))))

full-lists(s) := (s=nil) ? nil :
                append(full-list(first(s)),full-lists(but-first(s)))

elim-rep(l) := (l=nil) ? nil :
              member(first(l),but-first(l)) ?
              elim-rep(but-first(l)) :
              append(list(first(l)), elim-rep(but-first(l)))

```

The first class in the list, which overwrites the attribute, determines the default value.

Although CLOS and FAST follow completely different approaches for the determination of a default value, the results are isomorphic: As it can be seen in the formulas, both algorithms lead to the same class precedence list – apart from a reversal of the super-class lists. If a concept graph is given, and if the CLOS program extends the classes in the reverse order of the one used in the FAST program, then both programs lead to the same default value for an attribute. In CLOS, the first super-class is predominant, whereas in FAST, the last super-class is predominant. This results from a procedural interpretation of class extension in FAST. As the traversal used in FAST is isomorphic to the one used in CLOS, both algorithms satisfy the three ontological principles mentioned above.

3.3.4. Chapter Summary

This chapter has explored the conflicts resulting from multiple inheritance. Concerning the problem of ambiguous inherited attributes, CLOS, Java and FAST are guided by completely different assumptions about its meaning. As a result, such an attribute exists once in CLOS, twice in FAST and either once or twice in Java. Concerning multiply inherited attributes, CLOS and FAST behave the same way. Both languages implement a multiply inherited attribute once. Java does not know multiply inherited class attributes, because it does not support multiple inheritance with classes. This is the reason, why the inference problem with a multiply inherited attribute has only been programmed in CLOS and FAST.

The inference problem with a multiply inherited attribute has no universal solution. It has been pointed out that the only thing a programming language can do is searching for a value in the ancestor graph. The decision for a class is quite arbitrary, although certain guiding principles seem plausible. These principles include the unity of families, the preference of a sub-concept and a hierarchy among the direct super-concepts. It has been shown that both CLOS and FAST respect these principles. In terms of sub-concept layers, CLOS needs exponential time to determine a default value, whereas FAST only requires constant time. It should be noted that the inference problem could also be solved by hand: The conflicting attribute can be overwritten in the child class with a value. As constant and shared fields can be accessed without an instance in FAST, this allows for example an explicit choice for a value of a certain super-class⁴⁴.

⁴⁴ Sowa noted that this might be reasonable in some cases (2000).

4. Final Reflections

This part will summarize the results of this paper. CLOS, Java and FAST have been compared with respect to the most basic tasks of ontological structure representation: Concepts have been defined, attributes have been added, instances have been created, sub-concepts have been introduced and multiple inheritance has been investigated. Throughout the analysis, differences among the languages concerning their powerfulness of design, their expressiveness and their view of objects became evident. This chapter will sum up the usefulness of each of the languages for the representation of ontological structures. Then, an outlook will be given about topics, which would have exceeded the frame of this paper.

4.1. Conclusion

Throughout the investigation, it became apparent, that none of the languages is designed perfectly for the representation of ontological structures. First, they are all attached very closely to the computational model of Ontology. The drawbacks of this simplified form of the ontological model have already been pointed out. These include the assumption of discrete entities, the assumption of well-defined attributes and the extensional view of concepts. Furthermore, the investigation revealed that the programming languages tend to have a procedural view of knowledge rather than a declarative one. Consequently, their only inference mechanism is inheritance. Moreover, the variability of attribute values disallows proofs and implications in the strict sense of classical first order logic. Apart from these fundamental restrictions, neither CLOS nor Java nor FAST fulfilled all quality criteria at the same time. Each of them has its specific drawbacks in its powerfulness of design, its expressiveness or its view of objects. All of them failed in the issue of type refinement, for example. These observations might explain why object-oriented programming languages have not been used very often as a means of knowledge representation. On the other hand, programming languages allow integrating ontological structures efficiently into software systems, if one is willing to renounce certain demands. The usefulness of CLOS, Java and FAST for this purpose differs in certain aspects.

CLOS proved to be the most powerful language for the representation of ontological structures. Apart from the access to shared attributes and verified type refinement, all ontological tasks could be implemented. CLOS allows abstraction with all kinds of attributes and it supports multiple inheritance. Furthermore, it offers a high flexibility of method declarations, as it supports variable instance methods and multi-methods. The latter are a unique feature of CLOS. However, this powerfulness of design goes along with a very limited view of objects: Instances are seen as tuples of slots rather than independent individuals. Despite of accessor methods, there is no way to hide private fields or methods from external accesses in CLOS, so that encapsulation in a strict sense is not possible. Furthermore, CLOS does not encourage the use of constructors. This is also a drawback in data abstraction. These observations lead to the conclusion that objects are rather data structures than self-governing entities in CLOS, which makes the language less adequate for Ontology. If this aspect is considered less important by the modeler, CLOS convinces by its powerful object-oriented facilities.

Java showed a very advanced view of objects. It is the only language of the ones examined, which provides a sophisticated visibility policy. Fields, methods and even classes can be hidden from external access. Together with the support for constructors, this grants a high degree of complexity and independence to objects. Thus, the view of objects in Java meets the demands of Ontology. On the other hand, Java showed some severe disadvantages in its powerfulness of design. First, it neither allows multi-methods nor variable instance methods. If such a method is needed, it has to be simulated by complicated language constructs. Furthermore, Java does not support field abstraction. The shadowing of fields is not only an inconvenience in terms of expressiveness, but also a serious problem for the implementation of inference algorithms. In addition, the `static` attributes provided by Java cannot fulfill the task of shared attributes in an ontological structure. Moreover, Java does not support multiple inheritance. As it has been pointed out, interfaces proved to be a less useful solution for this purpose. Despite of its complex view of objects, Java lacks powerfulness of design for the representation of ontological structures.

FAST proved to offer a high expressiveness. Classes, methods and fields integrate coherently into the language. This syntactic uniformity guarantees a high degree of data abstraction. While a functional view of attribute accesses predominates in CLOS and an object-oriented one in Java, both views are possible in FAST. Most importantly, FAST programs are significantly less wordy than their

counterparts in CLOS and Java: In FAST, only 154 tokens⁴⁵ are needed for the implementations of this paper – compared to 244 in CLOS and 314, i.e. twice as many, in Java. Although these numbers only provide a spot check, they indicate the more straightforward programming style of FAST. Nearly all presented issues could be implemented in FAST. The language supports instance methods as well as field abstraction and multiple inheritance. Furthermore, FAST – unlike CLOS – performs all attribute accesses in constant time. However, it does not allow multi-methods as does CLOS. FAST shares a complex view of objects: There is a concept of privacy and support for constructors. Although objects are seen as self-responsible entities, FAST lacks the deepened policy of visibility of Java. In summary, FAST shares nearly the powerfulness of CLOS and nearly the view of objects of Java, while at the same time showing the highest degree of expressiveness.

4.2 Outlook

The present investigation does not include meta-concepts, i.e. concepts the instances of which are concepts. Meta-concepts are a powerful means of knowledge modeling. In the ontology, they allow reasoning about concepts as well as expressing collective properties. CLOS is the only language among the ones examined, which supports meta-concepts. These concepts translate into meta-classes, which are governed by a *meta-object protocol*. Apart from their immediate ontological use, meta-classes even allow configuring the inheritance mechanism. It can be specified, which attributes are inherited, which inheritance strategy is to be used and which attributes can be accessed. Thus, meta-classes multiply the facilities of CLOS, but the exact specification of the meta-object protocol has not yet been approved by the ANSI⁴⁶.

Furthermore, some object-oriented languages – among them Java and CLOS – allow objects to inspect properties of other objects and classes. This capability is called *reflection*. An object can for example find out which methods are defined in a class and it can even call them. Reflection permits an advanced independence of objects and a more flexible behavior at run-time. Reflection is closely related to other dynamic capabilities: It can for instance be permitted that a class definition be changed at run-time. CLOS supports this feature, whereas Java and FAST do not. A change of the class definition can be necessary if new properties of the concept become known at run-time. Moreover, it can be possible that an instance needs to change its class membership, because it does not fulfill the concept conditions any more. CLOS and FAST permit such a change, whereas Java does not. In summary, many other useful facilities are offered by the programming languages, which could not be detailed in this paper. It remains open whether these facilities will ever allow programming languages to represent reality.

⁴⁵ I counted identifiers and all syntactical elements such as brackets, parenthesis, semicolons, commas, quotes and dots. I considered those program fragments throughout the chapters, which exist in all three languages.

⁴⁶ See (Kopp, 1996) for an application of the meta-object protocol in another LISP dialect.

Appendix A. Program Code

This appendix contains the complete program code of this paper in CLOS, Java and FAST. The code fragments have been rearranged to form a proper program. The test calls to methods have been written in the language-dependent form, i.e. as functions in CLOS, as the `main` method in Java and as plain code in FAST.

Program Code in CLOS

```

; All code is in the file tiger.lisp

; ----- The class Antelope (Section 2.2.2.) -----
(defun Antelope ()
  ()
)

; ----- The class Tiger (Section 2.1.1.) -----
(defun Tiger ()
  (
    ; An attribute (Sections 2.1.1. and 2.3.2.)
    (name :initarg :name)

    ; A shared attribute (Section 2.1.2.)
    (dangerous :allocation :class :initform 100)

    ; A private attribute with accessors (Sections 2.1.3. and 2.2.4.)
    (hungry :accessor hungry)

    ; An attribute with a default value (Section 3.1.2.)
    (age :initform 10)
  )
)

; A method (Section 2.2.1.)
(defmethod roar ((a Tiger))
  (print "Rooaar!")
)

; A multi-method (Section 2.2.2.)
(defgeneric meet (a b))

(defmethod meet ( (a Tiger) (b Tiger) )
  (print "Tiger meets tiger")
)

(defmethod meet ( (a Tiger) (b Antelope) )
  (print "Tiger eats antelope")
)

(defun testTiger ()
  ; Creation of an object (Section 2.3.2.)
  (setq shirkan (make-instance 'Tiger :name "Shirkan"))
)

```

```

; A variable instance method (Section 2.2.3.)
; (This declaration has to follow the declaration of the instance)
(defmethod roar ((a (eql shirkan)))
  (print "Rooooooooaaaaaar!"))
)

; Use of accessor methods (Section 2.2.4.)
((setf hungry) t shirkan)
(print (hungry shirkan))

; Access to an attribute (Section 2.3.3.)
(print (slot-value shirkan 'name))

; Access to a shared attribute (Section 2.3.4.)
(print (slot-value shirkan 'dangerous))

; Call to a method (Section 2.3.5.)
(roar shirkan)
)

; ----- The class PetTiger (Section 3.1.1.) -----
(defun PetTiger (Tiger)
  (
    ; Introduction of a new attribute (Section 3.1.1.)
    (owner :initform "A")

    ; Overwriting a field (Section 3.1.3)
    (age :initform 0)

    ; Overwriting a shared field (Section 3.1.4.)
    (dangerous :initform 10)
  )
)

; Overwriting a method (Section 3.1.5.)
(defmethod roar ((a PetTiger))
  (print "mmmrurr")
)

(defun testPetTiger ()
  ; Creation of a PetTiger
  (setq kitty (make-instance 'PetTiger :name "Kitty"))

  ; Access to an overwritten shared field (Section 3.1.6.)
  (print (slot-value kitty 'dangerous))
)

; ----- The class CircusTiger (Section 3.2.1.) -----
(defun CircusTiger (Tiger)
  (
    ; Introduction of a new field (Section 3.2.1.)
    (owner :initform "B")
  )
)

```

```

    ; Overwriting a shared field (Section 3.2.1.)
    (dangerous :initform 50)

)
)

(defmethod stunt ((a CircusTiger))
  (print "Performing stunt")
)

; ----- The class PetCircusTiger (Section 3.2.2.) -----

(defun PetCircusTiger (PetTiger CircusTiger)
  ()
)

(defun testPetCircusTiger ()

  ; Creation of a PetCircusTiger
  (setq shirkitty (make-instance 'PetCircusTiger :name
                                "Shirkitty"))

  ; Access to an ambiguous inherited field (Section 3.3.1.)
  (print (slot-value shirkitty 'owner))

  ; Access to a multiply inherited field (Section 3.3.2.)
  (print (slot-value shirkitty 'name))

  ; Inference problem (Section 3.3.3.)
  (print (slot-value shirkitty 'dangerous))

)

; Calls to the test functions
(testTiger)
(testPetTiger)
(testPetCircusTiger)

```

Program Code in Java

```

// ----- The class Antelope (Section 2.2.2.) -----
// File: Antelope.java

class Antelope {
}

// ----- The class Tiger (Section 2.1.1.) -----
// File: Tiger.java

class Tiger {

  // An attribute (Section 2.1.1.)
  String name;

  // A shared attribute (Section 2.1.2.)
  static int dangerous=100;
}

```

```

// A "private" attribute with accessors (Sections 2.1.3. and 2.2.4.)
protected boolean hungry;
boolean getHungry() {
    return hungry;
}
void setHungry(boolean h) {
    hungry=h;
}

// A method (Section 2.2.1.)
void roar() {
    System.out.println("Rooaar!");
}

// A "multi-method" (Section 2.2.2.)
void meet(Object a) {
    if(a instanceof Tiger)
        System.out.println("Tiger meets tiger");
    if(a instanceof Antelope)
        System.out.println("Tiger eats antelope");
}

// A constructor (Section 2.3.2.)
Tiger(String n) {
    name=n;
}

// An attribute with a default value (Section 3.1.3.)
int age=10;

public static void main(String[] argv) {

    // Creation of an object (Section 2.3.2.)
    Tiger shirkan=new Tiger("Shirkan");

    // Access to an attribute (Section 2.3.3.)
    System.out.println(shirkan.name);

    // Access to a shared attribute (Section 2.3.4.)
    System.out.println(Tiger.dangerous);

    // Call to a method (Section 2.3.5.)
    shirkan.roar();

    // Use of accessor methods (Section 2.2.4.)
    shirkan.setHungry(true);
    System.out.println(shirkan.hungry);
}
}

// ----- The class PetTiger (Section 3.1.1.) -----
// File: PetTiger.java

class PetTiger extends Tiger {

    // Introduction of a new attribute (Section 3.1.1.)
    String owner;

    // Shadowing of a field, not useful here (Section 3.1.3.)
    // int age=0;
}

```

```

// Shadowing of a shared field (Section 3.1.4.)
static int dangerous=10;

// Overwriting a method (Section 3.1.5.)
void roar() {
    System.out.println("mmmmrrr");
}

// A constructor (Section 3.1.1.)
PetTiger(String n) {
    super(n);
    // Setting the new default value correctly (Section 3.1.3.)
    age=0;
}

public static void main(String[] argv) {

    PetTiger kitty=new PetTiger("Kitty");

    // Access to an overwritten shared field (Section 3.1.6.)
    System.out.println(kitty.dangerous);

}
}

// ----- The interface StuntPerforming (Section 3.2.4.) -----
// File: StuntPerforming.java

interface StuntPerforming {

    // Introduction of a constant shared field (Section 3.2.4.)
    static int dangerous=50;

    // Declaration of a new method (Section 3.2.4.)
    void stunt();

    // Declaration of a new field by accessor methods (Section 3.2.4.)
    String getOwner();
    void setOwner(String o);

}

// ----- The class PetCircusTiger (Section 3.2.5.) -----
// File: PetCircusTiger.java

class PetCircusTiger extends PetTiger implements StuntPerforming {

    // Implementation of the required method (Section 3.2.5.)
    public void stunt() {
        System.out.println("Performing stunt");
    }

    // Implementation of the required accessor methods (Section 3.2.5)
    public String getOwner() {
        return owner;
    }
    public void setOwner(String o) {
        owner=o;
    }

    // A constructor (Section 3.2.5)

```

```

PetCircusTiger(String n) {
    super(n);
}

public static void main(String argv[]) {

    // Creation of a PetCircusTiger
    PetCircusTiger shirkitty=new PetCircusTiger("Shirkitty");

    // Access to an ambiguous inherited field (Section 3.3.1.)
    System.out.println(((PetTiger)shirkitty).dangerous);
    System.out.println(((StuntPerforming)shirkitty).dangerous);

    // Access to a "multiply inherited field" (Section 3.3.2.)
    System.out.println(shirkitty.name);

}
}

```

Program Code in FAST

```

// All code is in the file tiger.fst

// ----- The class Antelope (Section 2.2.2.) -----
CLASS Antelope (
    EXTENDS object
)

// ----- The class Tiger (Sections 2.1.1. and 3.2.2.) -----
VCLASS Tiger n string (

    // Extension of object for "multi-method" (Section 2.2.2.)
    EXTENDS object

    // An attribute (Sections 2.1.1. and 2.3.2.)
    VAR name n

    // A shared attribute (Section 2.1.2.)
    SHARED dangerous 100

    // An attribute with "accessors" (Sections 2.1.3. and 2.2.4.)
    VAR hungry false

    // A method (Section 2.2.1.)
    FUNCTION roar (
        PRINT "Rooaar!"
    )

    // A "multi-method" (Section 2.2.2.)
    FUNCTION meet a object (
        IF (isa a Tiger) (
            PRINT "Tiger meets tiger"
        )
        IF (isa a Antelope) (
            PRINT "Tiger eats antelope"
        )
    )

```

```

    )
  )

  // A variable instance method (alternative declaration) (Section 2.2.3.)
  VAR roar2 (LAMBDA (
    PRINT "Rooaar!"
  ))

  // An attribute with a default value (Section 3.1.2.)
  VAR age 10
)

// Creation of an object (Section 2.3.2.)
VAR shirkan (Tiger "Shirkan")

// Access to an attribute (Section 2.3.3.)
PRINT (name shirkan)
PRINT (shirkan NAME)

// Access to a shared attribute (Section 2.3.4.)
PRINTINT (dangerous Tiger)

// Call to a method (Section 2.3.5.)
ROAR shirkan

// Redefinition of variable instance method (Section 2.2.3.)
LET (LAMBDA roar2 shirkan) (LAMBDA this Tiger (
  PRINT "Roooooaaaaaar!"
))

// Use of accessor methods (Section 2.2.4.)
LEThungry shirkan true
PRINTBOOLEAN (hungry shirkan)

// ----- The class PetTiger (Section 3.1.1.) -----
CLASS PetTiger n string (
  EXTENDS Tiger n

  // Introduction of a new attribute (Section 3.1.1.)
  VAR owner "A"

  // Overwriting a field (Section 3.1.3.)
  VAR age 0

  // Overwriting a shared field (Section 3.1.4.)
  SHARED dangerous 10

  // Overwriting a method (Section 3.1.5.)
  FUNCTION roar (
    PRINT "mmrrrr"
  )
)

// Access to an overwritten shared field (Section 3.1.6.)
PRINTINT (dangerous PetTiger)

```

```

// ----- The class CircusTiger (Section 3.2.1.) -----
CLASS CircusTiger n string (
  EXTENDS Tiger n

  // Introduction of a new field (Section 3.2.1.)
  VAR owner "B"

  // Overwriting a shared field (Section 3.2.1.)
  SHARED dangerous 50

  // Introduction of a new method (Section 3.2.1.)
  FUNCTION stunt (
    PRINT "Performing stunt"
  )
)

// ----- The class PetCircusTiger (Section 3.2.2.) -----

CLASS PetCircusTiger n string (
  EXTENDS PetTiger n
  EXTENDS CircusTiger n
)

// Creation of a PetCircusTiger
VAR shirkitty (PetCircusTiger "Shirkitty")

// Access to an ambiguous inherited field (Section 3.3.1.)
PRINT (owner (shirkitty : PetTiger))
PRINT (owner (shirkitty : CircusTiger))

// Access to a multiply inherited field (Section 3.3.2.)
PRINT (name shirkitty)

// Inference problem (Section 3.3.3.)
PRINTINT (dangerous shirkitty)

```

Appendix B. Bases of Relations

Definition: A *path* in a binary relation $R \subset A \times A$ is a sequence of elements of A so that each pair of an element and its immediate successor is an element of R .

Definition: A *basis* of a binary relation R is a binary relation R' with a minimal number of elements so that the transitive closure of R' is R .

Theorem: The basis of an acyclic binary relation is unique.

Proof: Let R be an acyclic relation. Let R' and R'' be two distinct bases of R . As the bases are distinct, it follows without a loss of generality that there exist distinct a and b so that $R'(a, b)$ and $\neg R''(a, b)$. From $R'(a, b)$, it follows that $R(a, b)$. From $R(a, b)$ and $\neg R''(a, b)$, it follows that there must be c so that $R(a, c)$ and $R(c, b)$. As a consequence, there must be two paths $p_1 = [a, x_1, \dots, x_n, c]$ and $p_2 = [c, y_1, \dots, y_m, b]$ in R' . p_1 may not contain b , because this would allow a cycle with b and c . Analogously, p_2 may not contain a . Then the element (a, b) of R' is superfluous, because $R(a, b)$ appears by joining the paths p_1 and p_2 . Thus, R' is not minimal and cannot be a basis.

Appendix C. References

- America, P. (1991). "A Behavioural Approach to Subtyping in Object-Oriented Programming Languages". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.
- Attardi, G. (1991). "An Analysis of Taxonomic Reasoning". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.
- Bateman, J. A. (1993). "Ontology construction and natural language". Visited 20 July 2003. <<http://www.darmstadt.gmd.de/publish/komet/papers/ontology-ws.ps>>
- Bench-Capon, T. J. M. (1990). "Knowledge representation: an approach to artificial intelligence". London: Academic Press
- Bibel, W., Hölldobler, S. and Schaub, T. (1993). "Wissensrepräsentation und Inferenz: eine grundlegende Einführung". Braunschweig, Germany: Vieweg
- Bracha, G. and Cook, W. (1990). "Mixin-Based Inheritance". Visited 9 September 2003. <<http://java.sun.com/people/gbracha/oopsla90.ps>>
- Brachmann, R. and Schmolze, J. G. (1985). "An Overview of the KL-ONE Knowledge Representation System". In Walz, D. L. (1985). "Cognitive Science, Volume 9". Norwood, NJ, USA: Ablex Publishing Corp.
- Cyphers, D. S. (1990). "Optimizations in the Symbolics CLOS Implementation". Visited 8 August 2003. <<http://www.apl.jhu.edu/~hall/text/Papers/CLOS-Optimizations.text>>
- Ducourneau, R. and Habib, M. (1991). "Masking and Conflicts, or To Inherit is Not to Own!" In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.
- Eysenck, M. W. and Keane, M. T. (2000). "Cognitive Psychology: A student's Handbook". London: Psychology Press
- Gat, E. (1999). "Lisp as an Alternative to Java". Jet Propulsion Laboratory, California Institute of Technology. Visited 10 August 2003. <<http://www.racai.ro/~trausan/LispJava.htm>>
- Gosling, J. (1995). "Java: an Overview". Visited 10 August 2003. <<http://java.sun.com/people/jag/OriginalJavaWhitepaper.pdf>>
- Gosling, J. et al. (2000). "The Java Language Specification, Second edition". Visited 10 August 2003. <http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html>
- Graham, P. (2001). "Five questions about language design". Visited 6 September 2003. <<http://www.paulgraham.com/langdes.html>>
- Guarino, N. (1998). "Formal Ontology and Information systems". In Guarino, N. ed. (1998). "Formal Ontology in Information Systems". Amsterdam: IOS press
- Holmevik, J. R. (1995). "The History of Simula". Center for Technology and Society, University of Trondheim. Visited 9 September 2003. <<http://java.sun.com/people/jag/SimulaHistory.html>>
- Horty, J. F. (1991). "A credulous Theory of Mixed Inheritance". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.
- Jones, D. M. and Paton, R. C. (1998). "Some Problems in the Formal Representation of Hierarchical Knowledge". In Guarino, N. ed. (1998). "Formal Ontology in Information Systems". Amsterdam: IOS press

- Keene, S. E. (1989). "Object-oriented programming in COMMON LISP: a programmer's guide to CLOS". Boston, MA, USA: Addison-Wesley
- Kopp, J. (1996). "Konstruktion von Wissensrepräsentationssprachen durch Nutzen und Erweitern objektorientierter Sprachmittel". Diss. Bielefeld U, 1995. Sankt Augustin, Germany: Infix
- Lyons, J. (1977). "Semantics". Cambridge: Cambridge Univ. Press
- Marek, V. W. and Truszczyński, M. (1993). "Nonmonotonic logic: context-dependent reasoning". Berlin: Springer
- Merriam-Webster, Inc. (1994). "Merriam Webster's Collegiate Dictionary, tenth edition". Springfield, MA, USA: Merriam-Webster, Inc.
- Minelli, A. (1993). "Biological systematics: the state of the art". London: Chapman & Hall
- Minsky, M. (1975). "A Framework for representing knowledge". In Brachmann, R. J. (1989). "Readings in knowledge representation". San Mateo, CA, USA: Morgan Kaufmann
- Padgham, L. (1991). "A Lattice-based Model for Inheritance Reasoning". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.
- Quillian, M. R. (1968). "Semantic Memory". In Minsky, M. ed. (1968). "Semantic Information Processing". Cambridge: MIT Press.
- Reed, S. L. and Lenat, D. B. (2002). "Mapping Ontologies into Cyc". Cycorp, Inc. Visited 22 July 2003. <http://www.cyc.com/doc/white_papers/mapping-ontologies-into-cyc_v31.pdf>
- ROMANS project (2003). "About AROM". Visited 9 September 2003. <<http://www.inrialpes.fr/romans/arom>>
- Rosch, E. (1978). "Principles of categorization". In Rosch, E. ed. (1978). "Cognition and categorization". Hillsdale, NJ, USA: Erlbaum
- Schneider-Patel, P. F. (1991). "What's Inheritance got to do with Knowledge Representation". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.
- Slade, S. (1998). "Object-oriented common LISP". Upper Saddle River, NJ, USA: Prentice Hall
- Smith, B. (1998). "Basic Concepts of Formal Ontology". In Guarino, N. ed. (1998). "Formal Ontology in Information Systems". Amsterdam: IOS press
- Snyder, A. (1991). "Inheritance in Object-oriented Programming". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.
- Sowa, J. F. (2000). "Knowledge representation: logical, philosophical, and computational foundations". Pacific Grove, CA, USA: Brooks/Cole
- Steele, G. L. (1990). "Common Lisp the Language, 2nd edition". Woburn, MA, USA: Digital Press. Available at <<http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html>>
- Steele, G. L. and Gabriel, R. P. (1993). "The Evolution of LISP". Visited 9 September 2003. <<http://www.cs.umbc.edu/331/resources/papers/Evolution-of-Lisp.pdf>>
- Stein, L. A. (1991a). "Computing Skeptical Inheritance". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.

Stein, L. A. (1991b). "A Unified Methodology for Object-oriented Programming". In Lenzerini, M., Nardi, D. and Simi, M. eds. (1991). "Inheritance hierarchies in knowledge representation and programming languages". New York: Wiley and Sons Ltd.

Suchanek, F. M. (2003). "FAST – A New Programming Language". Visited 1 August 2003. <<http://www-lehre.informatik.uni-osnabrueck.de/~fsuchane/fast.htm>>

Varzi, A. C. (1998). "Basic Problems of Mereotopology". In Guarino, N. ed. (1998). "Formal Ontology in Information Systems". Amsterdam: IOS press

Weiermann, S. L. (2000). "Semantische Netze und Begriffsdeskription in der Wissensrepräsentation". Diss. Salzburg U, 1999. Göppingen, Germany: Kümmerle

Winograd, T. and Bobrow, D. G. (1977). "An Overview of KRL, A Knowledge Representation Language". In Brachmann, R. J. (1989). "Readings in knowledge representation". San Mateo, CA, USA: Morgan Kaufmann

Erklärung

Hiermit erkläre ich, Fabian M. Suchanek, die vorliegende Arbeit "Representing Ontological Structures in CLOS, Java and FAST" selbstständig verfasst zu haben und keine anderen Quellen oder Hilfsmittel als die angegebenen verwendet zu haben.

Osnabrück, den 21.9.2003

Fabian M. Suchanek
Mat.Nr.: 900899