CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abteilung Software Engineering

Diploma Thesis

Ontology-based Metadata for MidArch-Styles

Reiner Jung

7th May 2008

Advisor:	Simon Giesecke
First examiner:	Wilhelm Hasselbring
Second examiner:	Simon Giesecke

Contents

1 Introduction

The development of software systems today is a difficult thing to do. During the past decades the project size and complexity have grown extensively and now vast set of goals have to be achieved in projects while various constraints must be adhered to. These constraints are often induced by legacy systems which have to be integrated into the new system or migrated to a new platform.

A common method in software development in is to divide software in smaller chunks which are easier to handle. The fabric that holds all these chunks together, the big picture, is called an software architecture.

Software architectures describe how components and connectors are arranged in software systems. Today many of these software systems share common characteristics and therefore their architectures share common characteristics too. So it is logical to put these characteristics down in a special document called architecture pattern or if they are more formalized architecture styles.

Architectural styles can represent generic styles [Monroe et al. 1997] or specialized, technology induced MidArch styles [Giesecke et al. 2007]. They describe specific characteristics of architectures, such as structural, topological, and behavioral characteristics. Also they contain information on data types, functional and non-functional properties [Metha 2004], like reliability, usability, or maintainability, as well as meta-properties which describe the meaning or usage of an architectural style.

Architectural styles make design concepts explicit so they can be shared between software engineers. A major problem with architectural styles is, that their building blocks are not easily set in relation to each other. This means component types which belong to different styles, but have a similar purpose are named differently. Also component types of different styles with the same name, can represent totally different concepts. So the relationship of styles have to be determined by the engineer herself. This is not a big problem when there are only a view styles, however with middleware-oriented architectural styles, this is quite different, because they can be numerous. So the software engineer has to analyze many style descriptions and compare them by hand.

Another problem of today's style description is, that there is a big conceptual gap between the description of styles and the way goals are described. While goals and goal-derived system properties address specific needs of the customer, styles address the needs of a system architect.

Therefore architectural styles today have two limitations in their concept. First, the comparability of the functionality of architectural style elements is limited. And second, the meaning of the elements in an architectural style is not defined or at least not set into relationship to other elements.

Based on these limitations, I derive at two questions:

- How can properties of elements of an architectural style be modeled?
- How can properties of different styles be made comparable with one another?

One way to address these two issues is to form a common and yet extendable vocabulary for meta-properties (and of course functional and non-functional properties as long as they are not part of the ADL approach used in MidArch [Giesecke et al. 2007]) of architectural style elements in a language which is more formal than a spoken language and is able to express relationships. One potential language for this purpose is the *Web Ontology Language* (OWL) [McGuinness and van Harmelen 2004] an extension to the *Resource Description Framework* (RDF) [Beckett 1999], which is capable of describing elements as objects with properties, and is comparable to first order logic and the Z notation [Spivey 1998].

The goal of my thesis is to test this approach by developing methods and tools to generate and use such vocabulary. Therefore I create a method to retrieve meta-information on architectural styles and store this information in an OWL based knowledge base. In the same way I develop a method for querying this knowledge base.

The thesis has the following outline. Chapter 2 discusses the foundations of my thesis. This comprises the definition of terms, the introduction of concepts, technologies, and software components. Chapter 3 discusses the requirements of the knowledge base, explains different solutions for these requirements, and finally defines the basic structures of the knowledge base. Chapter 4 discusses a technique to extract information from technical documentation and engineers and store it in the knowledge base. The querying technique, the counterpart to the modeling and storage technique is explained in chapter 5. The two techniques are then evaluated by analyzing a view middleware-oriented architectural styles, which have been defined and used in [Bornhold 2006]. This information is used in chapter 6 for evaluation of the introduced techniques. Finally, chapter 7 discusses the results of the evaluation and gives a glimpse on the future use of meta-information in software development.

2 Foundations

The primary goal of my thesis is to make information about middleware-oriented architectural styles comparable and accessible for a wide range of software architects. Therefore I develop a method to model meta properties of architectural styles, and describe a usable access method to the knowledge stored in these properties. In this chapter, I introduce and discuss basic concepts and technologies to describe and access knowledge.

Section 2.1 discusses the philosophical and linguistic terminology used in my thesis. Section 2.2 introduces the *Semantic Web* and its technologies, in which ontologies play a great role. In section 2.3, an ontology-based approach for architectural style modeling is explained. Section 2.4 is dedicated to the MidArch Method and its style modeling and selection procedures, because this method is the context for my work. Section 2.5, discusses two projects with different knowledge modeling approaches which make use of OWL-based ontologies to describe their knowledge. And finally in section 2.6, software frameworks, programs, and libraries are described, which could be used to implement of my method.

2.1 Disambiguation

This work is on meta-information for middleware-oriented architectural styles. The meta-information is formulated with the help of knowledge-management concepts. To understand these concepts, philosophical and linguistic terms have to be explained and distinguished from each other. This is especially important because these terms have different meanings in the sciences and arts.

The following text are therefor subdivided in three subsections. In section 2.1.1, I discuss the terms *data*, *information*, and *knowledge*, because they are the building blocks to understand the subsequent definitions. Section 2.1.2 is dedicated to the term *ontology*, because ontologies play a great role in my approach. At last, linguistic terminology is introduced in section 2.1.3. They are important to the knowledge retrieval process.

2.1.1 Data, Information and Knowledge

Data, information, knowledge, and ontology are philosophical terms and go back to definitions from Aristoteles and Platon [Krämer 1999, Schnelle 1976, Mittelstraß 1980a,b].

The term *data* refers to values like numbers, words, images, or any other result of an observation of variables. However data itself has no meaning or relation [FOLDOC 2007] it is just a set of values without an interpretation. For example *five* is a value or *green*. Five can be the number of apples in a basket or the age of a tree.

To interpret data, the values have to be correlated. These relations are also represented by data values. For the interpretability of data a system must be able to process data as symbols. Through correlation, data gets a meaning and becomes information.

The term *information* is defined in information theory as: Any configuration of symbols, which can be distinguished, identified, and created by a particular system, and where the system is able to transform the configuration into other configurations understood by the system [Schnelle 1976]. A symbol in this context is any data value which can be distinguished from other values.

A more general description of information shows that information can be understood as the form and representation of thought [Krämer 1999], which has an inter subjective property.

While information is an abstract and inter subjective representation of thought, knowledge is a subjective term. *Knowledge* involves a person or thing to be aware of information and its implications [Mittelstraß 1980b]. Using a strict definition of knowledge, machines were not able to know things. However knowledge can also be seen as information combined with pragmatics [Possner 1998, p. 515], which means, knowledge is interpreted information.

In the context of knowledge management, knowledge can be implicit (tacit knowledge) or explicit [Polanyi 1966]. Tacit knowledge is knowledge which is known by a person and where the person can act on, but the knowledge is not available to other persons. Explicit knowledge is therefore knowledge which has been expressed, so that other people can incorporate it into their knowledge. By the strict definition of knowledge, explicit knowledge is nothing more than information, because knowledge only exists when someone or something is aware of the information.

In this work, I use the term knowledge for interpreted or interpretable information and distinguish it from pure codified information. Thus they stand for different views on information. Where knowledge refers to the use of information and information itself refers to the codification and representation of information.

Based on this, the expression *isles of knowledge* an be explained. Isles of knowledge in this context are sets of information which are disjoint. So someone or something would not be able to infer information from all sets to gain new knowledge. Such situation is unfortunate, because it hinders the gain of knowledge. Therefor isles of knowledge should be interconnected by adding additional information. Also while developing knowledge management systems, the creation of such isles should be avoided.

2.1.2 Ontology

The term *ontology* is borrowed from philosophy [Mittelstraß 1980a], where it is a theory of *the study of being or existence.* The theory tries to describe basic categories and relationships which define entities and and types of entities. In computer science, ontology is defined as an explicit specification of a conceptualization. A conceptualization is an abstract and simplified view of the world for a specific purpose [Gruber 1993] like a model.

An ontology in computer science is a set of classes, relationships, objects, and properties, which describe a domain of knowledge and its vocabulary. Figure 2.1 shows an example with a class *fish* and two entities of this class *salmon* and *shark*. The entity *salmon* has specific characteristics or properties, like *born in fresh water*, *migrate to ocean to mature*, or *have a distinct color called salmon pink*. These properties differentiate the entity from the other entity *shark*.

This example already embodies a small hierarchical, tree like, structure of fish with two entities or objects in it. Such hierarchical structures are called *taxonomies*. The two entities in the example's taxonomy represent sets or classes of fish, because there is more than one species called shark (see figure 2.1).

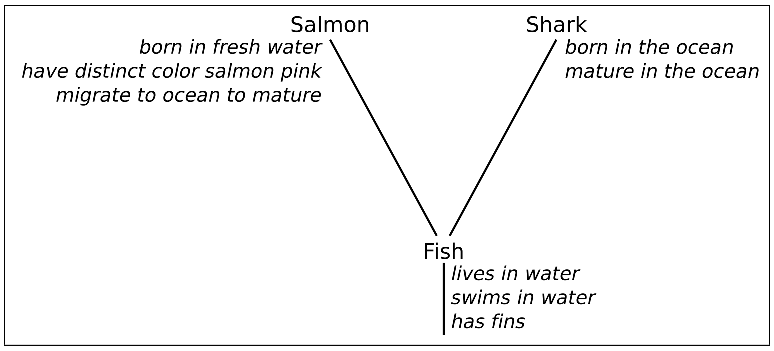


Figure 2.1: A small example taxonomy for the fish example with properties (in italics).

The fish taxonomy example already shows relationships to other entities. The salmon for instance is born in *fresh water*. Fresh water describes bodies of water with low concentration of salt and other dissolved matter, as found in lakes and streams. The entity salmon has therefore a property which refers to entities in a water ontology.

The property *has distinct color* of salmon refers to a color named *salmon pink*. Colors are normally not categorized in a taxonomy, but they can have several different relationships. Some of them can be ordered by their physical properties, like wavelength. Colors can also have a complementary color, for example red with its complementary color cyan.

The color example shows that not all entities or sets of entities can be represent by a taxonomy in a sensible way. Nevertheless, the colors have relationships and therefore they can be represent by an ontology. This is an important point, because ontologies allow a wide range of relationship patterns, but are often confused or limited to taxonomies, because taxonomies are so widely used to represent knowledge in a given domain.

2.1.3 Linguistic Terminology

This thesis discusses techniques (see chapter 4 and 5) which transform technical documentation into logic-based knowledge models like ontologies, rules, or clauses. While the previous sections explained the terms information, knowledge, and ontology, which are important to understand the modeling and use of meta-information. This section discusses the linguistic terminology used in my thesis.

First, I explain the basic elements of sentences. Second, I briefly describe the concept of negation in language. Third, I introduce the term conjunction. And finally, I discus the terms phrase, clause, and sentence, because they are used in content analysis and a clear understanding of the differences is important.

Subject, Predicate, and Object

The terms *subject* and *object* describe entities in sentences. The subject in a sentence is the origin of an action indicated by the predicate. The *predicate* defines the action to be performed. It describes what the subject is doing. Objects, however, are the recipient of the action, involved in the action itself, or they are the possessor of the subject [Haegeman and Guéron 1999]. Linguistics define even more object types represented by different cases, but for this thesis, these three types are sufficient.

The recipient of an action is called the *indirect object* and corresponds in English with the dative case [Haegeman and Guéron 1999, p. 129]. The object involved in the action is called the *direct object* and it corresponds in English with the accusative case [Haegeman and Guéron 1999, p. 75, p. 124, p. 128].

For example in the sentence *The server sends a message to the client.* The subject is *the server* indicating who is doing something. The action is expressed by the predicate *sends.* The direct object is *a message*, because it is the thing involved in the action. And the recipient of the action is *the client*, which is the indirect object.

The possessive object uses in English the genitive case [Haegeman and Guéron 1999, p. 412]. It indicates that the subject belongs to a possessor. For example *The application's identity server grants permissions.* The possessor in this sentence is the *application* which has a *server.* The *permissions* are obviously the direct object.

Negation

Beside the basic elements subject, predicate, and object, sentences may contain *negations* [Givón 1993, p. 187]. Negations are quite simple. They just negate a statement. For example *The server sends **no** message.* The word *no* indicates in this example that the server sends *not a message.* Because no alternative is given, the context suggests that the server is sending nothing at all. However the word *no* inverts the meaning of the object not the predicate.

A negation on the predicate is done with *not* which is also used in concatenations like, *cannot* [Givón 1993, p. 187]. For example: *The client cannot connect to the server.* In this example the word *cannot* inverts the predicate *connect* indicating the action is not possible.

Beside *no* and *not*, there are other negative structures in English which are depend on the logical structure of the text. A detailed discussion of negative structures can be found in [Swan 2005, p. 344–350,352–356].

Conjunction

The last relevant part of speech used in this thesis are *conjunctions* [Givón 1993, p. 78]. Conjunctions, like *and*, *but*, and *or*, are words which connect words, phrases or clauses. The meaning of *and* is to define sets of entities or actions. For example: *The server sends and receives messages.* or *The server provides sounds and images.*

The conjunction *or* mostly defines alternatives. For example: *The server sends or receives messages.* Compared to the example with *and*, there is a important difference. The server is either sending messages *or* receiving messages. So the *or* indicates an exclusive alternative. The sentence with *and*, however, allows to conclude that the server is doing both [Givón 1993, p. 78f].

The conjunction *but* is normally used to join two clauses, where the second clause contradicts the first one. Also it is used to exclude entities or actions. For example: *The server sends but never receives messages.*

Phrase, Clause, and Sentence

In the previous paragraphs, the terms phrase, clause, and sentence have been used, but not defined explicitly. The term *phrase* is defined as a group of words which represent a single syntactic unit (non-terminal) [Haegeman and Guéron 1999, p. 64]. There are different types of phrases which represent different syntactic units, like subject, object, or predicate. Subject and object phrases are also called *noun phrases*, because they have similar properties.

A *clause* consists of one or more phrases to formulate a complete or incomplete thought [Haegeman and Guéron 1999, p. 22]. A complete thought is represented by an independent clause for example *The server sends a message.* An incomplete thought is represented by a dependent clause, like *because it received an request.*

A *sentence* is build with clauses. The simplest sentence consists of one independent clause. More complex structures are achieved with conjunctions, which allow the conjunction of independent clauses. These structures are called compound sentences. Additionally an independent clause can be combined with one or more dependent clauses to construct complex sentences [Haegeman and Guéron 1999, p. 23]. And finally, compound and complex sentences can be mixed to get even more complex structures.

An understanding of these structures is necessary in content analysis, which is used in chapter 4 and 5 for the decomposition of information and information retrieval.

2.2 The Semantic Web

The *Semantic Web* was introduced by Berners-Lee et al. [2001] as an extension to the existing World Wide Web. The idea is to make data and information, which are right now bound to applications, available to agents and a variety of programs, so that these programs can combine information from different sources and can infer answers for user specified questions. To allow multiple programs to interpret a multiplicity of data the data and the rules to interpret them have to be expressed in a common set of languages and technologies as shown in figure 2.2.

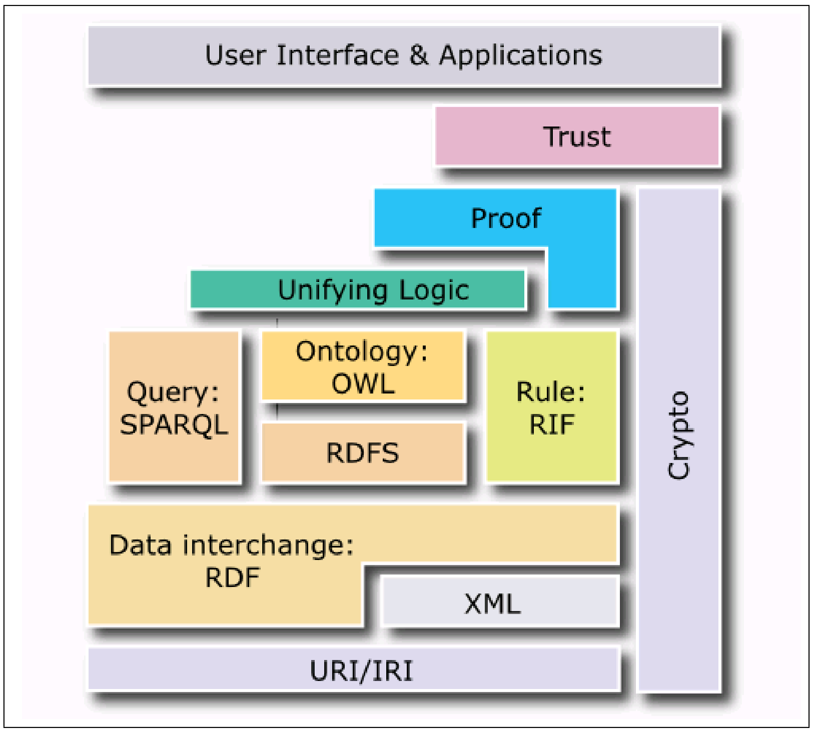


Figure 2.2: Technology stack of the Semantic Web [W3C 2007]

The *Semantic Web Initiative* developed in the past seven years languages and technologies to form the Semantic Web. The basis are classic web technologies like *Unified Resource Identifier* (URI) to identify entities and XML as one possible syntactic representation of data. On that foundation the initiative defined the Resource Description Framework which allows simple subject-predicate-object structures (clauses 2.1.3) to be expressed. Based on that the Web Ontology Language (OWL) and a rule language are defined, to express information and rules about their interpretation. Currently different approaches for query languages and their properties [RIF-WG 2008] are discussed. As a preliminary result, the RDF query language, SPARQL was defined, as well as the SWRL/RuleML language.

The layers *Unifying Logic*, *Proof*, and *Trust* are currently not subject to standardization. However, some work has already been done, and a small set of usable reasoners exist. They cover the layers Unifying Logic and Proof, where Trust is subject to a different standardization effort.

The following subsections explain the basic technologies of the Semantic Web technology stack. Starting with RDF, as a basis notation, followed by OWL, and the rule language specification SWRL.

2.2.1 Resource Description Framework

The *Resource Description Framework* (RDF) comprises several specification documents, including documents about the abstract syntax of RDF, the serialization of RDF data in XML, the semantics of RDF, and their description language. In most cases, the term RDF is used to address the RDF/XML syntax specification [McBride 2004] which describes the grammar of RDF as well as its XML representation. In this thesis I refer to RDF as the abstract syntax defined in [McBride 2004], because the abstract syntax allows to describe all relevant aspects in a compact manner. Also different representations of RDF can be derived from the abstract syntax.

RDF describes information in triples of subject, predicate, and object. A set of such triples form an RDF graph, like the example in figure 2.3. The subject and object elements are visualized as ellipse or boxes where the predicate symbolized by an arrow usually called an arc starting at the subject and pointing toward the object. The subject of a triple is either an URI, a literal, or a blank node. The URI refers to any possible resource on the Internet. A literal is a simple text string. And a blank node, is a node without URI or literal name, but it is still identified as a unique node. This means all blank nodes are distinguishable.

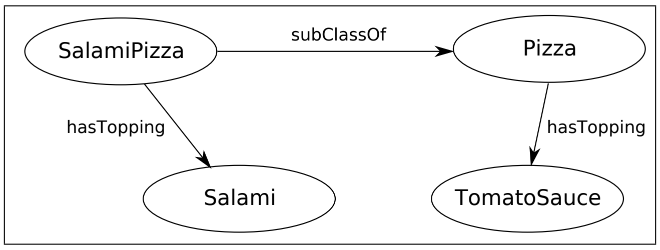


Figure 2.3: A simple RDF-graph about Italian cuisine.

The second central specification off the Resource Description Framework is the *RDF Vocabulary Description Language* [Brickley and Guha 2004] also referred to as *RDF Schema* (RDFS). Where RDF specifies the syntax of RDF graphs, RDFS specifies the semantic of such RDF graphs. The example RDF-graph in figure 2.3 shows four ellipses representing objects and subjects as well as three arcs representing predicates. In RDF the predicates do not have any meaning and only the knowledge of a human being allows the conclusion that `SalamiPizza` has both toppings `TomatoSauce` and `Salami`.

A software system however cannot come to the same result, on the basis of RDF, because *subClassOf* has no meaning in RDF. The RDFS however defines such meanings. For instance the predicate *rdfs:subClassOf* is defined in RDFS as a property which expresses the inheritance of properties, between RDF entities. Through such semantic definitions, RDFS allows to build simple ontologies, which are used as a basis for the Web Ontology Language explained in the next section.

2.2.2 Web Ontology Language

The RDF Vocabulary Description Language is able to define simple ontologies, but its vocabulary is not expressive enough to create complex ontologies for knowledge bases. The *Web Ontology Language* (OWL) [McGuinness and van Harmelen 2004] is specifically suited for this task. It comes in three variants, called OWL Lite, OWL DL, and OWL Full. OWL Full represents the full language without any restrictions, where OWL DL is a sub-set of OWL Full and OWL Lite is a further sub-set of OWL DL for simpler purposes.

OWL Full has the greatest expressive power of all three OWL languages. Every valid RDF graph is also a valid OWL Full structure and vice versa. In OWL Full classes could be treated as individuals. This ability allows to build a taxonomy tree in a knowledge base of animals, where the leaves are concrete animal types which could also be understood as individuals.

However in another knowledge base such animal types can be used as a class for a set of real animals. For example, ”cat” is an individual animal race in the animal taxonomy, but in another knowledge base ”cat” stands for a class with some individuals called Tom, Kitty, or Hamlet. In addition to this double usage, OWL Full considers data values also to be part of the individual domain. This means the set of datatype properties and the set of object properties are not disjoint sets.

However, OWL Full expressiveness can lead to knowledge bases that cannot be handled with description logic reasoners[Baader et al. 2007]. Results from such knowledge bases might not be unique or complete. One solution to this problem of ambiguity is to place some restrictions on the OWL language to make it compatible with description logic.

OWL DL requires a pairwise separation between classes, data types, data type properties, object properties, annotation properties, ontology properties, individuals, data values and the built-in vocabulary. This has various implications. First, classes cannot be individuals like in OWL Full. Second, the set of object properties and datatype properties are disjoint. Third, datatype properties cannot be of the property types *inverse of*, *inverse functional*, *symmetric*, or *transitive*. Finally, cardinality constraints cannot be placed on transitive or inverse properties [Bechhofer et al.].

In comparison, while OWL Full allows the use of all RDFS vocabulary, OWL DL allows only some few RDFS terms, like `rdfs:range` and `rdfs:domain`. Axioms in OWL DL must be well-formed and form a tree-like structure. Additionally, axioms about individual equality and difference must be about named individuals only, which means, that unnamed nodes are not allowed.

The third OWL language is called OWL Lite. It is designed to suit purposes of migrating simple taxonomies to OWL. OWL Lite provides a minimal subset of OWL language features, which are relatively easy to be supported by applications. It is a simplified version of OWL DL and therefore inherits the restrictions of OWL DL. In addition, OWL Lite forbids the use of `owl:oneOf`, `owl:unionOf`, `owl:complementOf`, `owl:hasValue`, `owl:disjointWith`, and `owl:DataRange` which are widely used in more complex knowledge bases. This limits OWL Lite mostly to simple information structures.

There are also restrictions on the usage of several other OWL elements with the intention to ease tool development. For instance set intersections are only allowed on sets with more than one element and the elements of these sets must be class names or property restrictions. More detailed information on these restrictions can be found in [Bechhofer et al.] and are beyond the scope of this thesis.

For the purpose of this thesis, OWL DL is the best choice, because it is the most expressive language which still conforms to description logic [Baader et al. 2007]. OWL Full can lead to undecidable structures and OWL Lite is not expressive enough. Therefor I use OWL DL in this thesis.

OWL DL Semantic Structure

In the following, I explain OWL DL as the most suitable language for the Semantic Web. As an example, I am using the knowledge base of "Pizza" from [Horridge et al. 2004], because it serves as a very descriptive example.

OWL DL has an XML serialization of its syntax, but it has also a, so called, *abstract syntax notation* which is more compact and easier to read for humans. The abstract syntax does not have a structure for every element of the XML serialization, because some elements only exist to ease the development, but do not increase the expressiveness. To give a complete picture on OWL DL, I describe all examples in both notation.

I start the explanation with the term class and the composition of classes in OWL. Followed by the discussion of property description. At last I describe the term individual and the definition of individuals.

OWL DL Classes

The OWL term *class* does not correspond to the term class used in most object-oriented programming languages. An OWL class is better compared to a mathematical set. Such set could be defined by individuals enumerating the elements of the set, or by property restrictions defining constraints on individuals belonging to a class. Additionally these classes can be constructed with set operators, like union, intersection, and difference.

The basic building blocks of OWL classes are *class axioms* [Bechhofer et al.]. The simplest class axiom is the plain declaration of a class as shown in listing 2.1. The class `Pizza` from the listing does not express anything, because it is not populated by any individuals, nor associated with other classes or properties.

```
<owl:Class rdf:ID="Pizza"/>

Class(Pizza partial owl:Thing)
```

Listing 2.1: Definition of a simple class

Beside simple declaration, classes can also be defined by: sub-classing, enumerating their individuals, stating that a class is the equivalent or definitely the opposite of another class. In order to understand the definition of classes in OWL, it is necessary to explain their relationship to individuals.

In OWL, individuals belong to sets which are associated to classes. Such sets are called *class extensions.* This is important, because a class extension can be associated to different classes which are not directly related to each other. Also, all set operators in classes work with the individuals forming the class extension.

These operators and properties are called *class axioms.* An important class axiom, which is heavily used in taxonomies, is the RDFS property *rdfs:subClassOf.* It is used to define a class as a sub-set of another class. To be more precise the class extension of the sub-class is a sub-set of the parent class extension. It is possible to specify more than one class to be a parent of a class. This leads to a class extension which is the intersection of all class extensions of the parent classes.

The *owl:equivalentClass* construct allows to express that two classes have the same class extension, while their definition could be completely different from each other. This is useful to combine knowledge bases from different sources, where classes are defined with different viewpoints in mind, but describing the same concept.

The opposite of the *owl:equivalentClass* is *owl:disjontWith.* In OWL classes are not disjoint by default, which means it has to be expressed explicitly. This is called an *Open World Model* [Horridge et al. 2004]. Therefore two classes which are not related have to be separated from each other.

```
<owl:Class rdf:ID="Pizza">
   <owl:disjointWith rdf:resource="Drink"/>
</owl:class>

Class(Pizza partial owl:Thing)
DisjointClasses(Pizza Drink)
```

Listing 2.2: Disjoint classes

For instance, to separate drinks from pizza, the definition in listing 2.2 serves as an example. It separates all `Pizza` individuals from those who belong to `Drink`. If we now define `SalamiPizza` as a sub-class of `Pizza` the class extension of `SalamiPizza` will also be disjointed with `Drink`, because all individuals of `SalamiPizza` also belong to `Pizza`.

Until now, I explained only classes which can be composed from other classes. In the following, I describe how these classes get their meaning by populating their class extension or by applying restrictions to properties.

In listing 2.3 the class `Pizza` is described by an enumeration of three different elements. Asked about what kinds of pizza are known a system could answer with these three elements. If asked about the term `SalamiPizza` the system could state it is a `Pizza`.

```
<owl:Class rdf:ID="Pizza">
  <owl:oneOf rdf:parseType="Collection">
    <owl:Thing rdf:about="#MargerithaPizza"/>
    <owl:Thing rdf:about="#PepperoniPizza"/>
    <owl:Thing rdf:about="#SalamiPizza"/>
  </owl:oneOf>
</owl:Class>

Class(Pizza partial oneOf(
    MargerithaPizza
    PepperoniPizza
    SalamiPizza
))
```

Listing 2.3: Class defined by an enumeration of individuals

But there are more than three varieties of pizza and of course the system still does not know anything about the term or the structure of `Pizza`. Therefore the class needs to be associated with additional information. For this little example of Italian cuisine, a pizza is composed of a base and some toppings. These two properties could be expressed by object properties and restrictions applied to them.

OWL has two different types of property restrictions. One restricts the cardinality of the property the other restricts the set of values. A cardinality restriction could be a minimum, maximum or a fixed number. For instance there is only one pizza base for a concrete pizza (see listing 2.4).

```
<owl:Class rdf:ID="Pizza">
  <owl:Restriction>
     <owl:onProperty rdf:resource="#hasBase"/>
     <owl:cardinality
          rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
  </owl:Restriction>
  <owl:Restriction>
     <owl:onProperty rdf:resource="#hasTopping"/>
     <owl:allValuesFrom rdf:resource="#PizzaTopping"/>
  </owl:Restriction>
</owl:Class>

Class(Pizza partial
   restriction(hasBase cardinality(1))
   restriction(hasTopping allValuesFrom(PizzaTopping))
)
```

Listing 2.4: Class with a property restriction

A restrictions on value sets limits the range of a property. With *owl:hasValue* the restriction could be set to a specific value. The other two value restrictions are equivalent to the math quantifiers for sets ∀ (for all) and (∃ (exists) and named *owl:allValuesFrom* and *owl:someValuesFrom*. The *owl:allValuesFrom* restriction expresses that all values of the given property must be of the specified type or range. In my example all toppings must be of the type `PizzaTopping`. This excludes for instance `Drink` from the list of values, because they are not sub-classed from `PizzaTopping`.

Beside these basic description methods, OWL allows to compose classes with the set operators ⊔ (union), ⊓ (intersection), and ¬ (complement) named `owl:unionOf`, `owl:intersectionOf`, and `owl:complementOf`. These operators work on the instances of the class. That means, a class which is defined as an intersection of two other classes is defined by those individuals which belong to both defining classes. Analogous union means that the individuals of the defined class are instances of one of the two defining classes.

The complement defines the set of individuals of a class by negation. So all individuals which are not instances of the complemented class are instances of the defined class. In listing 2.5 the class `NonVegetarianPizza` is defined as the complement of `VegetarianPizza`. However, this definition is incomplete, because a non-vegetarian pizza should still be a pizza. The second definition in listing 2.5 is therefore a better description for non-vegetarian pizza, because it intersects the set of non-vegetarian individuals with the set of `Pizza` instances.

```
<owl:Class rdf:about="#NonVegetarianPizza">
  <owl:complementOf rdf:resource="#VegetarianPizza"/>
</owl:Class>
<owl:Class rdf:about="#NonVegetarianPizza">
  <owl:intersectionOf rdf:parseType="Collection">
     <owl:Class>
        <owl:complementOf rdf:resource="#VegetarianPizza"/>
     </owl:Class>
     <rdf:Description rdf:about="#Pizza"/>
  </owl:intersectionOf>
</owl:Class>

Class(NonVegetarianPizza complete
   complementOf(VegetarianPizza))
Class(NonVegetarianPizza complete
   intersectionOf(
      complementOf(VegetarianPizza)
      Pizza
))
```

Listing 2.5: Complement of a class

OWL DL Properties

Up to this point, classes and the methods of their composition have been explained. Now OWL property axioms are explained. First, the two classes of properties in OWL DL are explained. Second, inheritance of property characteristics with *rdfs:subPropertyOf* is described. Third, the axioms *domain* and *range* are discussed. Fourth, the definition of properties through equivalence and inversion with the axioms *equivalentProperty* and *inverseOf* are explained. Fifth, the axioms for the property characteristics *functional* and *inverse functional* and their relationship with mathematical relation types *surjective* and *injective* are discussed. Finally, the transitive and symmetric characteristics of properties and their corresponding axioms are explained.

OWL has two different types of properties, called datatype property and object property. Datatype properties link individuals to data values, while object properties link individuals to individuals [Bechhofer et al.]. A property can therefore be seen as a binary relation, where subjects are related to objects or values. Each element of such a relation is called an *instance of a property* and all instances of a property form the *property extension* of that property.

The definition of properties in OWL is based on the property concept of RDFS [Brickley and Guha 2004]. The two types `owl:DatatypeProperty` and `owl:ObjectProperty` are subclasses of `rdf:Property`. In OWL Full object and datatype properties are not disjoint. However, in OWL DL and Lite they are disjoint.

The simplest way to define a property is just by name. Listing 2.6 defines the property *hasBase* and states that the values of this property has to be individuals, because it is defined as an *object property*. Beside this declarative axiom for properties, RDF and OWL define nine other axioms.

```
<owl:ObjectProperty rdf:about="#hasBase"/>

ObjectProperty(hasBase)
```

Listing 2.6: Simple property axiom

The *rdfs:subPropertyOf* axiom allows to express that a property is a sub-property of another property. This implies that all instances of the sub-property also belong to the parent property. In listing 2.7 the property *hasBase* is defined as sub-property of *hasIngredient*. This implies that having a ”base” means also having an ”ingredient”.

The next two axioms define the *domain* and *range* of an property. A property is a relation between subjects and objects. The subjects are individuals and belong to a class extension specified by the domain.

While the domain specifies the subjects, the range specifies the objects of a property. As shown in listing 2.7 the individuals of the class `Pizza` have the property *hasBase* and the base must be an individual from `PizzaBase`. This ensures that a topping cannot be erroneously assigned to a pizza as a pizza base.

```
<owl:ObjectProperty rdf:about="#hasBase">
   <rdfs:subPropertyOf rdf:resource="#hasIngredient"/>
   <rdfs:range rdf:resource="#PizzaBase"/>
   <rdfs:domain rdf:resource="#Pizza"/>
   <owl:inverseOf rdf:resource="#isBaseOf"/>
</owl:ObjectProperty>

SubPropertyOf(hasIngredient
   ObjectProperty(hasBase
     inverseOf(isBaseOf)
     domain(Pizza) range(PizzaBase))
)
```

Listing 2.7: Property definition

In some cases it is useful to define a property as the inverse of another property. For example, a salami pizza might *have a* thin and crispy base. The inverse expression would be: the thin and crispy base *is the base of* a salami pizza. Formally, the instances of the property extension are inverted. So all subjects become objects and all objects become subjects. To state that a property is the inverse of another property the *inverseOf* property is used.

Another relational property axiom is *equivalentProperty*. This axiom states that the property has the same property extension as the referenced property. This does not imply that both properties are equivalent. They can still relay to two different concepts. To state that two properties are identical, the *sameAs* axiom must be used.

In some cases it must be guaranteed that there is only one object to a subject. Which means, an individual can only have one value for a specific property. For example there shall only be one "base" per pizza. This can be ensured by declaring the property to be *functional* (see listing 2.8). Such a functional property is a *surjective* relation between the domain and the range.

As shown in listing 2.8 a property can also be declared as *inverse functional*. This means, that the subject in the property relation is fully determined by the object. This corresponds with an *injective* relation between domain and range.

```
<owl:ObjectProperty rdf:about="#hasBase">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
   <rdfs:subPropertyOf rdf:resource="#hasIngredient"/>
   <rdfs:range rdf:resource="#PizzaBase"/>
   <rdfs:domain rdf:resource="#Pizza"/>
   <owl:inverseOf rdf:resource="#isBaseOf"/>
</owl:ObjectProperty>

SubPropertyOf(hasIngredient
   ObjectProperty(hasBase
      inverseOf(isBaseOf)
      Functional InverseFunctional
      domain(Pizza) range(PizzaBase))
)
```

Listing 2.8: Definition of a functional property

The last two property types describe transitive and symmetric properties. A transitive property defines that if a tuple (a, b) is an instance of the property and (b, c) is also an instance of the same property then (a, c) is also an instance of the property, even if it is not explicitly specified. For example, Oldenburg is in Germany and Germany is in Europe. Therefore Oldenburg is in Europe too.

A symmetric property describes when there is an instance (a, b) of the property then there is also an instance (b, a). For example if Alice is a friend Bob than Bob is also a friend of Alice.

OWL DL Individuals

The last aspect to explain of OWL is the definition and the handling of individuals. They can be defined by class membership, property values, and by identity.

In listing 2.9 the individual `Margaritha` is defined as member of the class `Pizza` with a `ThinAndCrispyBase` and three toppings. So this individual is defined by class membership and four properties.

```
<Pizza rdf:ID="#Margaritha">
  <hasBase rdf:resource="#ThinAndCrispyBase"/>
  <hasTopping rdf:resource="#MozzarellaCheese"/>
  <hasTopping rdf:resource="#TomatoTopping"/>
  <hasTopping rdf:resource="#OreganoTopping"/>
</Pizza>

Individual(Margaritha type (Pizza)
   value(hasBase ThinAndCrispyBase)
   value(hasTopping MozzarellaCheese)
   value(hasTopping TomatoTopping)
   value(hasTopping OreganoTopping)
)
```

Listing 2.9: Individual defined through class membership

The definition of individuals by identity is realized through three axioms: *owl:sameAs*, *owl:differentFrom*, and *owl:AllDifferent*. The axiom *owl:sameAs* is used to express that an individual is identical to another, like an alias. However, the individual may have additional property values.

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <owl:Thing rdf:about="#Margaritha" />
    <owl:Thing rdf:about="#Italian" />
    <owl:Thing rdf:about="#SalamiPizza" />
  </owl:distinctMembers>
</owl:AllDifferent>

DifferentIndividuals(Margaritha Italian SalamiPizza)
```

Listing 2.10: Individuals separated with `owl:AllDifferent` axiom

The axiom *owl:differentFrom* allows to state that two individuals are definitely not the same. This is important in ontologies with an open world model [Horridge et al. 2004] as it is the case with OWL ontologies. However, in ontologies with many individuals this could lead to a vast set of *owl:differentForm* statements to keep them all disjoint. Because this is an error-prone and costly approach, OWL provides the *owl:AllDifferent* construct to separate all individuals in a collection (see listing 2.10).

2.2.3 The Rule Language SWRL

The strength of OWL is the modeling of knowledge in form of ontologies based on classes, individuals, and properties. However, general rules about individuals, classes, and properties cannot be expressed in a straightforward way. For instance, an OWL-property called *hasParent* can be defined as a relation between humans. Furthermore the class *human* might have a cardinal restriction to that property of two humans.

Expressing that a human has tow biological parents. A property *hasSibling* would also be defined as a relation between humans. The problem is that for all human individuals the relationships have to be set explicitly, because OWL is not able to express rules.

Yet a rule language is able to express *hasSibling* as general rule instead of setting each relationship by hand. The *hasSibling* rule would state that two humans who have the same parent are siblings. So rules can be used to define knowledge of individuals which is hard to define in OWL. Additionally, any rule would apply to all individuals in OWL. Furthermore every time a new individual is added, the sibling relations have to be adapted.

But rules cannot only be used to define knowledge, they can also be used to formulate queries. OWL by itself has no querying capabilities. Sent to a reasoner, the program tries to determine additional information out of the given definitions, but to find information the ontology has to be browsed by hand. Therefore search engines utilizing OWL use an additional search language such as SPARQL [Prud'hommeaux and Seaborne 2008] to allow searches on OWL databases.

The simpler search languages just look for classes, individuals, or properties which match the expression given in the search request. However, they can only find individuals which are already there and are not able to conclude answers to a given question from the present knowledge. A more complex language is SPARQL [Prud'hommeaux and Seaborne 2008]. SPARQL works on RDF-triples, however it can only search RDF-graphs, which means it can only find defined information by looking for objects, which have certain relationships to other objects in the RDF-graph. Also SPARQL does not know of rules, because a rule language is not in the scope of RDF. Rule languages however can be used to formulate questions which then can be answered by deductions utilizing facts and rules from a knowledge base.

Different organizations and research groups have proposed such rule languages to supplement OWL with rules and rule based querying. In an effort to coordinate the rule language standardization process the W3C has initiated a working group, called *Rule Interchange Format* (RIF) Working Group. One goal of this working group is the development of a rule language called *Basic Logic Dialect* (RIF-BLD) [Boley and Kifer 2007]. The document about RIF-BLD is still a work in progress.

The language is composed of two sub-languages. One to formulate conditions, which could be used in rule bodies, the antecedent, or in queries [Boley and Kifer 2007]. The second sub-language is based on clauses with no more then one positive literal, called Horn clauses [Horn 1951]. The work on RIF-BLD looks quite promising, however, the language specification is still under development and no software exists which supports RIF-BLD or any sub-set of that language.

The National Research Council of Canada, Network Inference now Cerebra Inc., and Stanford University proposed a rule language called *Semantic Web Rule Language* (SWRL) [Horrocks et al. 2004]. It is not a language as powerful as RIF-BLD. SWRL is also not an official W3C standard, but it is widely supported by reasoners like Pellet (see section 2.6.2), Fact++, or Jess (see section 2.6.4) and ontology editors, like Protégé (see section 2.6.1). Therefore I use SWRL as rule language, which can be replaced by RIF-BLD in future, when the language is final and rule engines and reasoners exist. The language SWRL is based on RuleML [Hirtle et al. 2006] and allows therefor combine RuleML with the OWL DL sub-language.

The rules are disjunctions of literals with no more then one positive literal, which are called horn clauses [Horn 1951]. They can be written as an implication between an antecedent (body) and consequent (head), where both, antecedents and consequents, are a conjunction of atoms, called predicates. Because SWRL rules are based on Horn clauses, an empty antecedent implies 'true', while an empty consequent is interpreted as 'false'.

$$\bigwedge_{i=1}^{n} atom_i \rightarrow \bigwedge_{j=1}^{m} atom_j$$

Atoms in SWRL are predicates accepting one or more terms as parameter. The predicates represent OWL descriptions and properties, or predefined predicates, such as $sameAs(x, y)$ or $differentFrom(x, y)$.

OWL descriptions are predicates $C(x)$ with one parameter where C is the name of a class or a data range in OWL and x is an individual name or a value of a data range. The predicate holds iff x is an instance of C, or for data ranges iff x is a value of the data range C.

In OWL, properties are relations between a domain and a range. The domain is an OWL class and the range is an OWL class or a data range. The property predicate $P(x, y)$ has therefore two parameters, where x represents an individual from the domain and y an individual or a data value from the range of the property. The predicate holds iff x has the property y.

Besides the OWL classes and properties, SWRL has two predefined built-in predicates, called $sameAs(x, y)$ and $differentFrom(x, y)$. $sameAs(x, y)$ holds iff x is the same individual or data value as y and $differentFrom(x, y)$ holds iff x is not the same as y. Additionally to these two predicates, it is possible to extended a reasoner with other predicates, which can be used with the $builtin(r, x_1, \ldots, x_n)$ predicate. The parameter r is the name of such an built-in predicate and $x_1, \ldots x_n$ are the parameter for the built-in predicate.

The letters x and y in this description are either variables, OWL individuals or OWL data values. To distinguish variables from individuals and data values, SWRL prefixes variables with a question mark.

The specification of SWRL is given in an abstract syntax notation similar to the OWL abstract syntax notation. Additionally an XML representation of SWRL is defined for internal use of tools. And because XML is hard to read, a human readable form with $antecedent \rightarrow consequent$ is specified as well. Both antecedent and consequent are a conjunction of atoms. For example the rule

$$hasParent(?x, ?y) \land hasBrother(?y, ?z) \rightarrow hasUncle(?x, ?z)$$

expresses that any individual x which has a parent y and where that parent has a brother z has an uncle. And the uncle of the individual x is z.

Beside the additive nature of SWRL to OWL, there are also structures which can be expressed in both languages. In OWL, a property can be transitive or symmetric. These characteristics can also be expressed by rules. For instance, the property *isLocatedIn*(x, y) for geographic regions is transitive, which means if Oldenburg is located in Germany and Germany is located in Europe, then Oldenburg is located in Europe too. This transitive example is expressed in OWL as:

```
ObjectProperty(isLocatedIn Transitive domain(Region) range(Region))
```

or in SWRL as a rule:

$$\mathit{isLocatedIn}(?x, ?y) \wedge \mathit{isLocatedIn}(?y, ?z) \rightarrow \mathit{isLocatedIn}(?x, ?z)$$

The symmetric characteristic can be expressed in OWL with an object property that has the type *Symmetric*. In SWRL the same symmetric characteristic can be expressed by a rule i.e. $P(?x, ?y) \rightarrow P(?y, ?x)$.

Besides the transitive and symmetric characteristics, the relationships describing of similarity and difference can be expressed in SWRL with the predicates *sameAs* and *differentFrom* and in OWL with the properties `owl:sameAs` and `owl:differentFrom`. The two predicates can be seen as "syntactic sugar" [Horrocks et al. 2004], because they do not increase the expressiveness of the language. However, they are convenient for rule designers.

SWRL is a language to describe and define knowledge with rules. Such rules, could in theory, be used to formulate queries which can in return be used to infer new knowledge. However SWRL alone has no idioms to display the results. Therefore the Protégé-project developed a small addition to SWRL, called SQWRL [SQWRL 2007].

SQWRL is not an official or proposed standard, however it is a rule based way to formulate queries. SQWRL introduces a small set of predicates, which can be used in the consequent. SQWRL's predicates are influenced by SQL and provide the following features:

- *sqwrl:select* allows to output variables and variable combinations, similar to SQL query results.
- *sqwrl:count* counts the number of identical variable values in the result set. It works similar to the SQL `GROUP BY` statement.
- *sqwrl:avg*, *sqwrl:max*, and *sqwrl:min* calculates the average, the maximum, and minimum of variable values respectively if they are numerial.
- *sqwrl:orderBy* and *sqwrl:orderByDescending* sorts the results of a query in an ascending and descending way.
- *sqwrl:selectDistinct* allows to eliminate duplicate results.

Together with SWRL-based predicates, SQWRL can be used to create complex queries which is the case in the knowledge querying technique in chapter 5.

2.3 Ontology-based Architectural Style Modeling

Pahl et al. [2007] introduced an approach to model architectural styles as ontologies. Their goal is a rich yet easily extensible semantic style modeling language with operators to combine, compare, and derive architectural styles. Beside descriptions, which can be made in architecture description languages, they want to store information about maintainability, dependability and performance in the ontology. The ontology language they use is called $\mathcal{ALC}$ specified in [Baader et al. 2007]. $\mathcal{ALC}$ is, however, not used as a replacement for an ADL, instead it is used to augment existing ADLs [Pahl et al. 2007].

In subsection 2.3.1 the language basics of $\mathcal{ALC}$ is introduced and compared to OWL, because both languages are designed for ontologies. Subsection 2.3.2 introduces their respective basic ontology vocabularies. And subsection 2.3.3 explains how architectural styles and relationships between them are expressed with ontologies.

2.3.1 Description Logic Language $\mathcal{ALC}$

The ontology language $\mathcal{ALC}$ is a member of the family of attributive languages $\mathcal{AL}$ [Baader et al. 2007, p. 51]. The letters $\mathcal{AL}$ stand for *attributive language* and the the letter $\mathcal{C}$ indicates that $\mathcal{ALC}$ allows the unification of concepts and full existential quantification, where $\mathcal{AL}$ only allows this with the universal concept [Baader et al. 2007, p. 52].

$\mathcal{ALC}$ is a description logic language with similarities to OWL DL. It provides a set of operators and notational elements for ontologies. In [Pahl et al. 2007] $\mathcal{ALC}$ is used to define an ontology for architectural styles. In addition to the operators defined for $\mathcal{ALC}$, [Pahl et al. 2007] defines extra operators to apply restrictions to architectural styles or perform refinements on them.

$\mathcal{ALC}$ consists of three basic notational elements: *concepts*, *roles*, and *individuals* also called *nominals*. A *concept* is a set of objects, where all objects have the same properties (but not necessarily the same property values). This corresponds with the definition of *class extension* in OWL. *Roles* in $\mathcal{ALC}$ are relations of concepts to concepts. Their definition corresponds with OWL properties, which are relations between a domain and a range. The third type of notation element are *nominals* also called *individuals*. A nominal is a named element of a concept.

On a logical basis, *individuals* can be seen as values or constants, concepts as unary predicates and roles as binary predicates, just like their counterparts in OWL and SWRL. Therefore everything which can be expressed in OWL and SWRL can be expressed in $\mathcal{ALC}$. Table 2.1 shows the correspondents of OWL and $\mathcal{ALC}$ descriptions, data ranges, properties, individuals, and data values, where table 2.2 shows OWL DL axioms and facts.

The $\mathcal{ALC}$ language allows the composition of concepts through a set of operators. Let C and D be concepts in $\mathcal{ALC}$. The negation of a concept ($\neg C$), the disjunction or union of concepts ($C \sqcup D$), the intersection or conjunction of concepts ($C \sqcap D$), and the implication of concepts ($C \rightarrow D$) are also concepts.

OWL Abstract Syntax	**DL/$\mathcal{ALC}$ Syntax**
Descriptions(C)	
A	A
`owl:Thing`	$\top$
`owl:Nothing`	$\bot$
`intersectionOf(`$C_1 \dots C_n$`)`	$C_1 \sqcap \dots \sqcap C_n$
`unionOf(`$C_1 \dots C_n$`)`	$C_1 \sqcup \dots \sqcup C_n$
`complementOf(`C`)`	$\neg C$
`oneOf(`$o_1 \dots o_n$`)`	$\{o_1\} \sqcup \dots \sqcup \{o_n\}$
`restriction(`R `someValuesFrom(`C`))`	$\exists R.C$
`restriction(`R `allValuesFrom(`C`))`	$\forall R.C$
`restriction(`R `hasValue(`o`))`	$R : o$
`restriction(`R `minCardinality(`n`))`	$\geq nR$
`restriction(`R `maxCardinality(`n`))`	$\leq nR$
`restriction(`U `someValuesFrom(`C`))`	$\exists U.D$
`restriction(`U `allValuesFrom(`C`))`	$\forall U.D$
`restriction(`U `hasValue(`v`))`	$U : v$
`restriction(`U `minCardinality(`n`))`	$\geq nU$
`restriction(`U `maxCardinality(`n`))`	$\leq nU$
Data Ranges(D)	
D	D
`oneOf(`$v_1 \dots v_n$`)`	$\{v_1\} \sqcup \dots \sqcup \{v_n\}$
Object Properties(R)	
R	R
`inv(`R`)`	R^-
Datatype Properties(U)	
U	U
Individuals(o)	
o	o
Data Values(v)	
v	v

Table 2.1: Correspondents of OWL DL descriptions, data ranges, properties, individuals, and data values in abstract OWL syntax with description logic (DL) syntax [Baader et al. 2007, p. 474].

These set-theory-based operators alone are not sufficient to describe complex structures. Therefore $\mathcal{ALC}$ defines value restrictions and existential quantification. In OWL these terms are represented by property restrictions. A value restriction in $\mathcal{ALC}$ is expressed as: $\forall R.C$, where R is a role and C is a concept. This restricts the role's range to C. The existential quantification is expressed as $\exists R.C$, which forces the elements of

OWL Abstract Syntax	**DL/$\mathcal{ALC}$ Syntax**
`Class(`A `partial` $C_1 \ldots C_n$`)`	$A \sqsubseteq C_1 \sqcap \ldots \sqcap C_n$
`Class(`A `complete` $C_1 \ldots C_n$`)`	$A \equiv C_1 \sqcap \ldots \sqcap C_n$
`EnumeratedClass(`A $o_1 \ldots o_n$`)`	$A \equiv \{o_1\} \sqcup \ldots \sqcup \{o_n\}$
`SubClassOf(`C_1 C_2`)`	$C_1 \sqsubseteq C_2$
`EquivalentClasses(`$C_1 \ldots C_n$`)`	$C_1 \equiv \ldots \equiv C_n$
`DisjointClasses(`$C_1 \ldots C_n$`)`	$C_i \sqcap C_j \sqsubseteq \bot, i \neq j$
`Datatype(`D`)`	
`ObjectProperty(`R `super(`R_1`) ...super(`R_n`)`	$R \sqsubseteq R_i$
`domain(`C_1`) ...domain(`C_m`)`	$\geq 1R \sqsubseteq C_i$
`range(`C_1`) ...range(`C_l`)`	$\top \sqsubseteq \forall R.C_i$
`[inverseOf(`R_0`)]`	$R \equiv R_0^-$
`[Symmetric]`	$R \equiv R^-$
`[Functional]`	$\top \sqsubseteq \leq 1R$
`[InverseFunctional]`	$\top \sqsubseteq \leq 1R^-$
`[Transitive])`	$Tr(R)$
`SubPropertyOf(`R_1 R_2`)`	$R_1 \sqsubseteq R_2$
`EquivalentProperties(`$R_1 \ldots R_n$`)`	$R_1 \equiv \ldots \equiv R_n$
`DatatypeProperty(`U `super(`U_1`) ...super(`U_n`)`	$U \sqsubseteq U_i$
`domain(`C_1`) ...domain(`C_m`)`	$\geq 1U \sqsubseteq C_i$
`range(`D_1`) ...range(`D_l`)`	$\top \sqsubseteq \forall U.D_i$
`[Functional])`	$\top \sqsubseteq \leq 1U$
`SubPropertyOf(`U_1 U_2`)`	$U_1 \sqsubseteq U_2$
`EquivalentProperties(`$U_1 \ldots U_n$`)`	$U_1 \equiv \ldots \equiv U_n$
`Individual(`o `type(`C_1`) ...type(`C_n`)`	$o \in C_i$
`value(`$R_1 o_1$`) ...value(`$R_n o_n$`)`	$\langle o, o_i \rangle \in R_i$
`value(`$U_1 v_1$`) ...value(`$U_n v_n$`))`	$\langle o, v_i \rangle \in U_i$
`SameIndividual(`$o_1 \ldots o_n$`)`	$\{o_1\} \equiv \ldots \equiv \{o_n\}$
`DifferentIndividuals(`$o_1 \ldots o_n$`)`	$\{o_i\} \sqsubseteq \neq \{o_j\}, i \neq j$

Table 2.2: OWL DL axioms and facts in abstract syntax notation compared to description logic syntax used by $\mathcal{ALC}$ [Baader et al. 2007, p. 476].

the concept to have a value for role R of the range C. As these restrictions work on concepts, the result are also concepts.

Like OWL, $\mathcal{ALC}$ allows to build sub-classes of classes. This is done by indicating that a concept is a sub-concept of another concept $C \sqsubseteq D$. So C is a sub-concept of D. Analogous sub-properties in OWL are sub-roles in $\mathcal{ALC}$ expressed as $S \sqsubseteq R$.

2.3.2 Basic Architectural Style Ontology Vocabulary

After the definition of the $\mathcal{ALC}$ language basics in the previous section, the definition of the architectural style ontology vocabulary can be explained. The vocabulary consists of five major concepts: configuration, component, connector, role and port types. All five types are considered sub concepts of *ArchType* [Pahl et al. 2007, p. 4]. In addition to these concepts, the roles *hasPart*, *hasInterface*, and *hasEndpoint* are defined.

The concept *configuration type* is defined as a set of component, connector, role and port types with $Configuration \equiv \exists hasPart.(Component \sqcup Connector \sqcup Role \sqcup Port)$. The $\exists$ indicates that the concept must have component, connector, role and port types. The relation *hasPart* connects them to configuration types. Therefore the term *Configuration* represents the most generic architectural style.

The component type is defined as an architectural style element (*ArchType*) which has a port type assigned: $Component \equiv ArchType \sqcap \exists hasInterface.Port$. Analogue the connector type is defined as an *ArchType* which has a role type assigned. The role type of a connector type must not be confused with the concept named roles in $\mathcal{ALC}$. A $\mathcal{ALC}$ role is a relation between two concepts and the role type of a connector type is a property of the connector. A slight inaccuracy exists in the definition of the connector type in [Pahl et al. 2007], as the definition does not ensure that a connector needs to have two endpoints. However it can be assumed that the connector type has at least two endpoints. Otherwise it would not be able to connect anything.

The role *hasPart* is a relation of configuration types to component, connector, port, and role types. The *hasInterface* role is a relation of component types to port types. Further *hasEndpoint* is a relation of connector types to role types. These three roles are part of the basic vocabulary of the ontology and can be extended if necessary.

2.3.3 Definition of Architectural Styles

On the basis of the vocabulary introduced in section 2.3.2, architectural styles can be defined by extending the basic vocabulary of elementary architectural styles. To explain this I use the Pipes-and-Filter style as an example from [Pahl et al. 2007].

The Pipes-and-Filter style has three component types *DataSource*, *DataSink*, and *Filter*. They form the set of Pipes-and-Filter component types. Analogically, the style has a connector type called *Pipe*. In a formal way this can be described as:

$$\begin{aligned}
PipeFilterPort &\sqsubseteq Port \\
PipeFilterPort &\equiv (Output, Input) \\
DataSource &\equiv \leq 1 hasPort \sqcap \exists hasPort.Output \\
DataSink &\equiv \leq 1 hasPort \sqcap \exists hasPort.Input \\
Filter &\equiv \leq 2 hasPort \sqcap (\exists hasPort.Output \sqcup \exists hasPort.Input) \\
PipeFilterComponent &\equiv (DataSource, DataSink, Filter) \\
PipeAndFilterStyle &\sqsubseteq Configuration \\
PipeAndFilterStyle &\equiv \exists hasPart.(PipeFilterComponent \sqcup \\
&\quad PipeFilterConnector \sqcup PipeFilterPort \sqcup \\
&\quad PipeFilterRole)
\end{aligned}$$

Architectural styles and especially middleware-oriented styles are rarely used without modifications, restrictions or some refinement. The approach in [Pahl et al. 2007] therefore defines a set of special operators for architectural styles. The restriction, intersection and union operators in [Pahl et al. 2007] can be realized in OWL DL as well (see table 2.1). However, the refinement operator requires the assignment of properties to classes, which is not possible in OWL DL, because in OWL DL classes cannot be individuals.

2.4 MidArch Method

The MidArch method [Giesecke 2008, ch. 9] is designed to support software engineers in developing and migrating middleware-oriented software architectures. The method itself can be divided into the four activities definition, preparation, exploration, and implementation which can be split into several tasks as shown in figure 2.4.

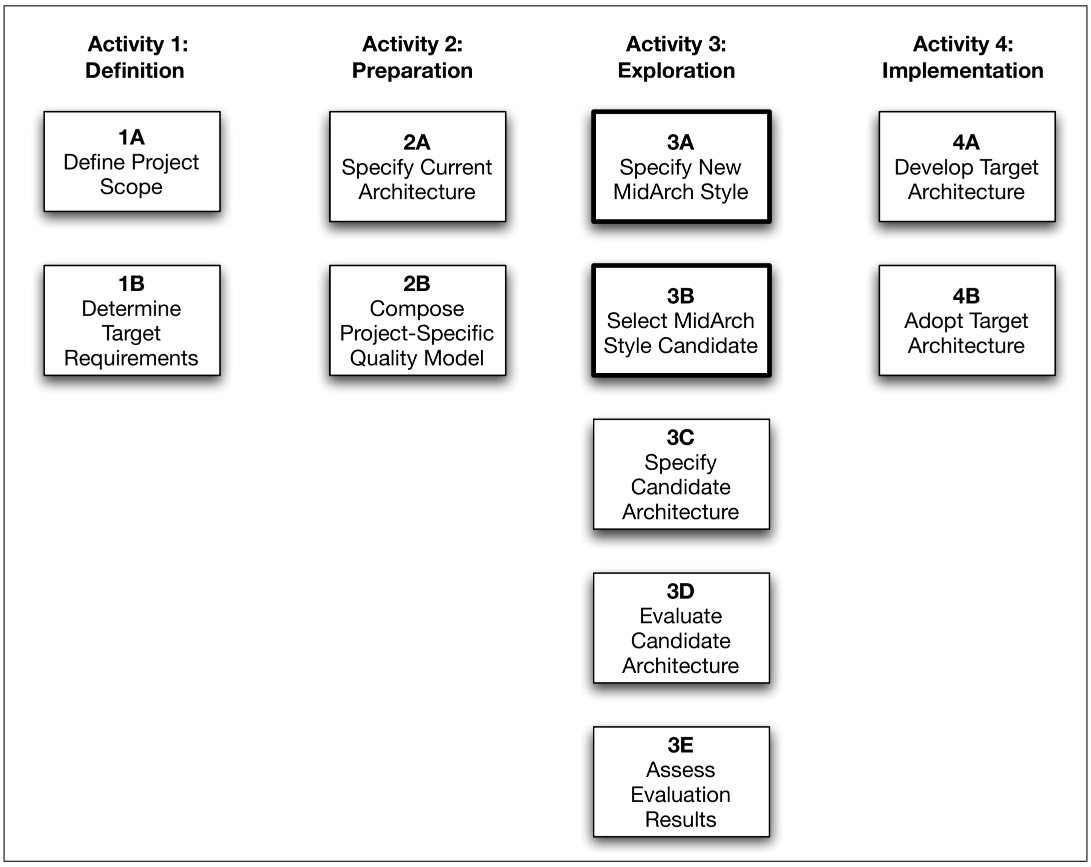

Activity 1: Definition
Activity 2: Preparation
Activity 3: Exploration
Activity 4: Implementation
1A Define Project Scope
1B Determine Target Requirements
2A Specify Current Architecture
2B Compose Project-Specific Quality Model
3A Specify New MidArch Style
3B Select MidArch Style Candidate
3C Specify Candidate Architecture
3D Evaluate Candidate Architecture
3E Assess Evaluation Results
4A Develop Target Architecture
4B Adopt Target Architecture

Figure 2.4: Overview of the MidArch method with its activities and tasks. [Giesecke 2008]

The *Definition* activity (activity 1) comprises of two tasks. First, the project scope is defined (task 1A) utilizing the current high-level system documentation. And second, the target requirements are determined (task 1B) on the basis of the project scope, which lead to a set of high-level requirements.

The activity 2, called *Preparation* activity, consists of two tasks. In task 2A the current architecture of the system, which is to be migrated is reconstructed based on the available information. If a new system is being built 2A is omitted. Task 2B is dedicated to the development of a quality model for the project. The quality model is based on the requirements defined in 1B.

Activity 3 is the central activity of the MidArch method. It is the *Exploration* activity with its five tasks (3A-3E). Task 3A is about forming a new MidArch style in an ADL for a given platform. The task can be repeated if more than one style has to be defined. In task 3B one or more existing MidArch styles are selected from a style repository. In task 3C the newly formed styles from task 3A or the selected styles from task 3B are used to create candidate architecture descriptions. These candidate architectures are evaluated in task 3D with the quality model from task 2B. All evaluation results are assessed in task 3E and finally an architecture is selected for activity 4.

Based on the selected architecture from task 3E and the current architecture from task 2A a final architecture is defined in task 4A. Finally, the architecture is implemented in task 4B utilizing a suitable architecture-based development method.

For my thesis the style modeling procedure from task 3A and style selection task 3B are of importance, because the collection of meta-information is similar to task 3A and the selection task might utilize meta-information in the selection process. Therefore the two tasks are discussed in the following two subsections 2.4.1 and 2.4.2.

2.4.1 Style Modeling Procedure

Giesecke [2008, ch. 8] introduces a five step procedure for style modeling. Figure 2.5 illustrates the procedure and the involved artefacts.

The initial step (1) produces an informal description of the style based on expert knowledge and the available documentation about the middleware product. The description covers all style-relevant aspects in natural language and yields a list of requirements. In the second step (2), relationships between the informal style description and general-purpose styles are identified. The *codification* step (3) maps the informal description to MidArch style modeling constructs as defined in [Giesecke 2008]. The codified informal style description and the relationships to MidArch styles are used together with an ADL to create a formal style description in the *formalization* step (4). In the *reviewing* step (5), the formal style is checked for consistency and probably altered to ensure consistency and instantiability.

2.4.2 Style Selection Procedure

In [Giesecke 2008, sec. 9.4] the selection of style candidates (task 3B) is briefly introduced. The selection process works on the MidArch style taxonomy with a project-

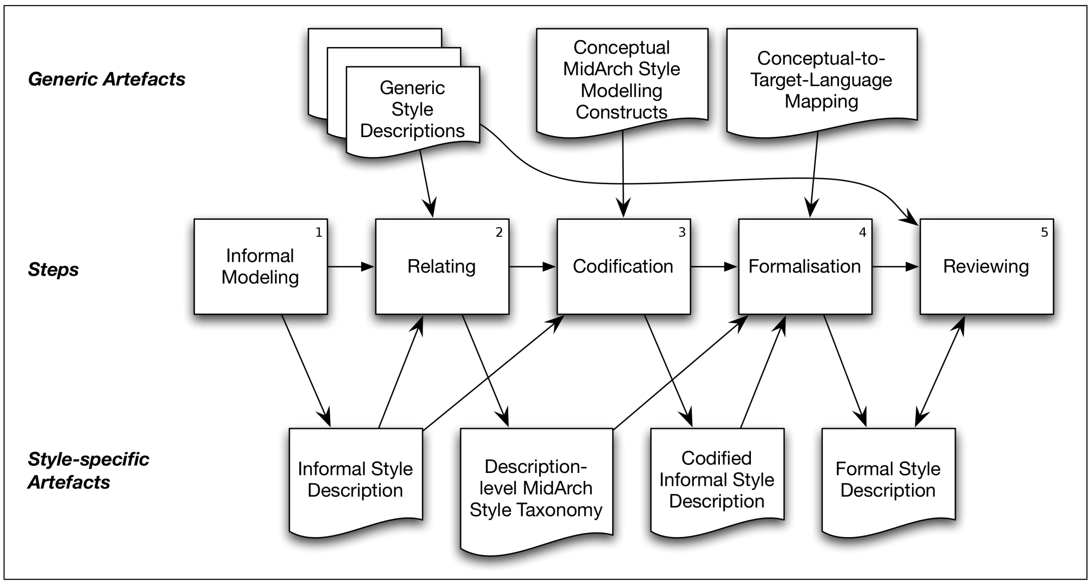

Generic Artefacts
Steps
Style-specific Artefacts
Generic Style Descriptions
Conceptual MidArch Style Modelling Constructs
Conceptual-to-Target-Language Mapping
1 Informal Modeling
2 Relating
3 Codification
4 Formalisation
5 Reviewing
Informal Style Description
Description-level MidArch Style Taxonomy
Codified Informal Style Description
Formal Style Description

Figure 2.5: General MidArch Style modeling method and artefacts involved [Giesecke 2008]

specific quality model and derived requirements. The styles in the taxonomy are evaluated MidArch styles from previous MidArch instances. The evaluation data is stored with the MidArch styles.

The selection process starts with a most generic style and proceeds hierarchically through the MidArch taxonomy. In this process, the evaluation data of every style is matched with the project-specific quality model. The results of these checks are ranked and the best matches are considered as style candidates.

2.5 OWL-based Knowledge Base Projects

The topic of my thesis is to create an ontology for meta-information about architectural styles. The development of an ontology is quite a complex process, because the domain, which is described by the ontology, must be analyzed carefully and entities have to be grouped. One problem is which of these entities are best represented by classes in OWL and which as individuals. While individuals have values or other individuals assigned to their properties, classes work with property restrictions to limit the number of possible values. This distinction of individuals and classes is especially important when OWL DL is used, because in OWL DL classes cannot be individuals like in OWL Full.

Instead of deducing an approach from scratch, I introduce two ontology projects, which use OWL DL. The first is the *Web of Pattern* (WOP) project and the second is about an *Urban Infrastructure Knowledge Base*. Both knowledge bases have to work around the described issue of individuals and classes.

2.5.1 The Web of Patterns Project

The *Web of Pattern* (WOP) is a toolkit for sharing knowledge about software design patterns. The toolkit comprises two parts: a RDF/OWL-based knowledge base and a set of Eclipse plug-ins. The toolkit can be used to detect patterns in Java code, create new patterns from Java code, rate patterns stored in the on-line database, and find relationships between patterns.

The ontology is formed around 13 classes, 17 object properties, and 11 data type properties. The patterns themselves are coded as individuals. The relationship between the patterns is implemented with the object property *isSubPatternOf*, which is a relation from pattern to pattern. This is quite different from the approach in [Pahl et al. 2007] where inheritance is expressed by sub-classing. The patterns in WOP are individuals, where the styles in $\mathcal{ALC}$ are represented by classes. Beside pattern descriptions, the ontology defines OWL classes to express concepts from the object oriented language domain. Therefore the concepts class, method, constructor, and field from that language domain are represented by OWL classes in WOP. The individuals of these OWL classes are representations of object oriented language structures. For instance, the Abstract Factory pattern is associated with the class *AbstractFactory* which is represented by an individual in WOP. In addition the constructors, methods, and fields are represented by individuals.

2.5.2 Urban Infrastructure Knowledge Base

The *knowledge base for urban infrastructure* [Osman and El-Diraby 2006] is designed to help share knowledge between different fields of urban infrastructure planning, deployment, and maintenance problems. The idea is to create an ontology which defines a vast set of concepts, which can be used in all fields to collect information about that field. Through the structure of the knowledge base the information is also available to ontology users from other fields.

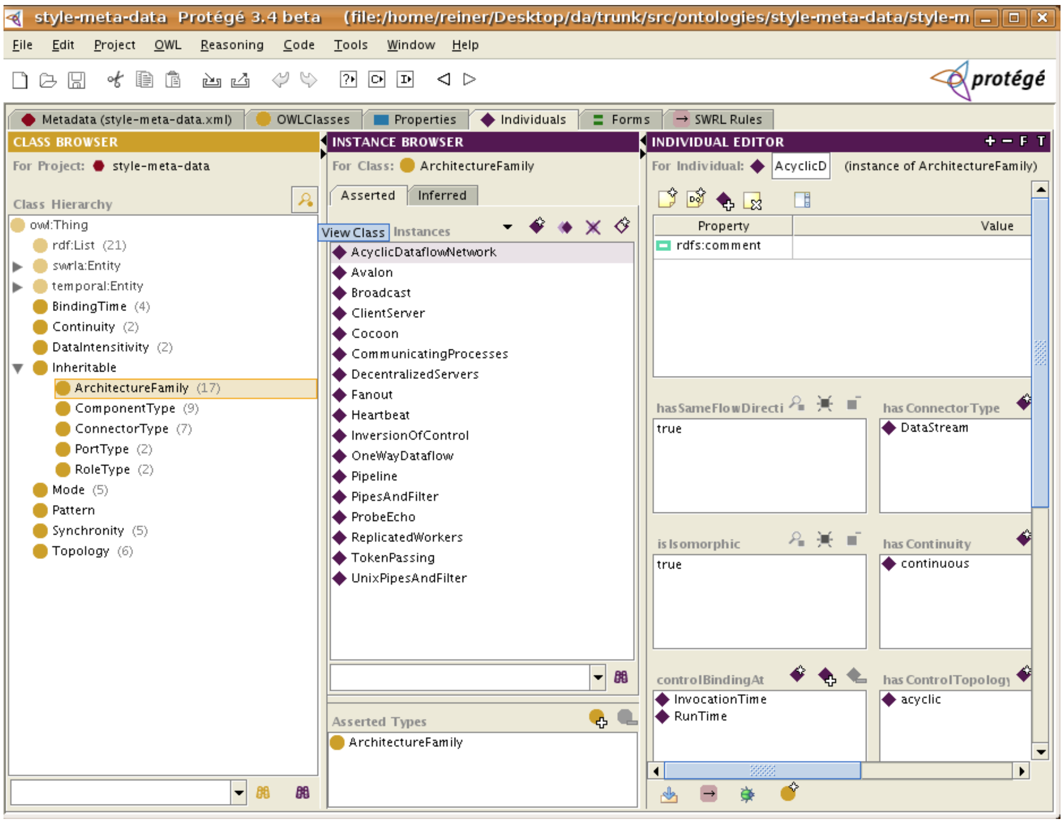
The ontology [Osman 2004] is a large set of classes and object properties with only view exemplary individuals. The purpose of this ontology is mainly to describe concepts and not concrete individuals. Therefor this ontology is more a class than an individual based knowledge base. As the primary goal is to share knowledge and allow browsing through the taxonomy, the class based approach is more suitable for this project, because the *subClassOf* relation has a defined semantic which is directly supported by the applications used in this infrastructure knowledge base.

2.6 Software Systems and Components

My thesis proposes a basic ontology for meta-information of architectural styles. Such a work cannot be done in reasonable time without proper tools. Further an ontology which cannot be queried is quite useless. Therefore other components are required. In the current section 2.6, I discuss different software components, applications and tools which can be used for outcome of my thesis.

2.6.1 Protégé-OWL

Protégé-OWL, from Stanford University, is one of the most popular OWL and RDF editor and development environments available today. It is based on the Protégé-platform, has a large supporting community and a wide user base. The editor is written in Java and can therefore run on many operating systems. It is under active development and many related projects create plug-ins and enhancements for Protégé-OWL. The sources of the project are published under GPL v2 [FSF 1991].


style-meta-data Protégé 3.4 beta (file:/home/reiner/Desktop/da/trunk/src/ontologies/style-meta-data/style-m
File Edit Project OWL Reasoning Code Tools Window Help
protégé
Metadata (style-meta-data.xml) OWLClasses Properties Individuals Forms SWRL Rules
CLASS BROWSER
INSTANCE BROWSER
INDIVIDUAL EDITOR
Asserted Inferred
View Class Instances
Property Value
Asserted Types

For Project: style-meta-data

Class Hierarchy
- owl:Thing
  - rdf:List (21)
  - swrla:Entity
  - temporal:Entity
  - BindingTime (4)
  - Continuity (2)
  - DataIntensitivity (2)
  - Inheritable
    - ArchitectureFamily (17)
    - ComponentType (9)
    - ConnectorType (7)
    - PortType (2)
    - RoleType (2)
  - Mode (5)
  - Pattern
  - Synchrony (5)
  - Topology (6)

For Class: ArchitectureFamily

- AcyclicDataflowNetwork
- Avalon
- Broadcast
- ClientServer
- Cocoon
- CommunicatingProcesses
- DecentralizedServers
- Fanout
- Heartbeat
- InversionOfControl
- OneWayDataflow
- Pipeline
- PipesAndFilter
- ProbeEcho
- ReplicatedWorkers
- TokenPassing
- UnixPipesAndFilter

Asserted Types: ArchitectureFamily

For Individual: AcyclicD (instance of ArchitectureFamily)

rdfs:comment

hasSameFlowDirecti	true
hasConnectorType	DataStream
isIsomorphic	true
hasContinuity	continuous
controlBindingAt	InvocationTime, RunTime
hasControlTopology	acyclic

Figure 2.6: Protégé's individual editor tab. On the left the class tree is shown, followed by the list of individuals for the selected class. The properties of the edited individual are on the right.

The core functionality of Protégé-OWL is the creation and manipulation of OWL and RDF knowledge bases. For this purpose it provides views, called tabs, for class, property, individual, and annotation management. The editor view for manipulating individuals can be seen in figure 2.6.

In addition to these editing capabilities the program integrates different reasoners by means of the DIG-interface standard [DIG 2006]. The reasoners CEL [CEL 2008], Fact++ [Tsarkov and Horrocks 2007], Pellet [Alford et al. 2007] and RacerPro [Racer

2008] provide support for the DIG-interface. The Pellet reasoner is also directly integrated into Protégé-OWL since version 3.4. The reasoners allow the user to check the consistency of the OWL knowledge base, compute inferred types of an ontology, classify the taxonomy, and perform a wide range of tests on the ontology.

Protégé-OWL does not only support OWL, it is also capable to support SWRL editing and in cooperation with Jess [Jess 2008] (see subsection 2.6.4) it is able to perform queries on a knowledge base with SQWRL.

The SWRL view allows the definition of rules on the basis of OWL properties. These rules symbolize either knowledge of their own, or, when SQWRL predicates are used in the consequent, queries about the knowledge. Jess itself is not able to process OWL ontologies directly, because it is a rule engine. Therefore Protégé-OWL has to transfer the knowledge for OWL and SWRL to a Jess-compatible rule format, and then feed the knowledge to the rule engine. After processing the rules Protégé-OWL converts the information back into OWL. All these features are controlled by logic behind the SWRLJessTab, a part of the SWRL view in Protégé-OWL.

The bridge between OWL and Jess in Protégé-OWL has, however, some limitations, because not all OWL axioms are transferred by the bridge. This is so, because newly inferred knowledge from Jess might be in conflict of present OWL knowledge. Yet these conflicts can be eliminated with a reasoner like Pellet [Protégé 2008].

A last notable feature of Protégé-OWL is its ability to function as a runtime environment for Protégé-based applications. Protégé-OWL has a form editor for this purpose. The defined forms together with the Protégé-runtime can be used as separate applications.

2.6.2 Pellet

Pellet is an OWL reasoner with support for SWRL and SPARQL, the RDF query language. The Pellet project was started at University of Maryland's Mindswap Lab and is now continued by Clark & Parsia. The reasoner is Open Source and is actively developed. Therefor any work based on Pellet can be used in future. Pellet also supports the upcoming OWL DL 1.1 standard which will increase the expressiveness of OWL DL without making it undecidable.

The Pellet reasoner supports the DIG-interface and works therefore with all frameworks and tools which support DIG, such as Protégé-OWL and Jena [Jena 2008], a Semantic Web framework. The reasoner supports querying on the basis of a subset of the SPARQL query language. However, SPARQL is only an RDF query language and does not support the semantics of OWL structures and SWRL rules. However, Pellet allows to augment OWL DL structures with SWRL rules, but has no rule based query interface at present. Therefore SWRL can only be used to describe general rules about individuals and enhance the inference of knowledge in the knowledge base. Therefore the query support must be realized with another program or component.

2.6.3 Bossam

Bossam is another reasoner package which supports OWL and SWRL input. It was developed at Electronics and Telecommunications Research Institute (ETRI) in Korea. The reasoner is freely available, however, its source code is not available. The last version is dated February 2007, the tool seems to be not actively under development. However it has some promising features and might become a useful replacement for Pellet if its development continues again.

Bossam is a reasoner for the Semantic Web and therefore supports OWL. In addition, it understands SWRL rules and comes with its own rule based language Buchingae. Buchingae is able to formulate more complex rules than SWRL and can also be used as a query language, however, it is not supported by any other platform and especially not by Protégé-OWL, which is the primary editor for my thesis. Another problem with Bossam is, a software bug with certain OWL documents generated by Protégé-OWL or other editors. Since the program is not available as source code and no new versions have been released since February 2007, Bossam is not used in this work.

2.6.4 Jess

Jess [Jess 2008] is a rule engine from the Sandia National Laboratories written in Java. It is used to develop and build expert systems. Jess cannot handle OWL classes or SWRL rules directly; they have to be converted into Jess rule language constructs. The Jess engine is not Open Source software, but it is freely available for academic research. The project is actively maintained and developed.

Jess has its own set of Eclipse [Eclipse 2008] plug-ins which can be used for rule development. Additionally, Jess has been integrated into Protégé-OWL and can therefor be used through an extension called SWRLTab. This extension handles the transfer of knowledge from OWL and SWRL to Jess. It also provides a query interface on basis of SQWRL. The SWRLTab allows to formulate queries in Jess language, which makes it a perfect tool for the knowledge query task.

The Jess language [Friedman-Hill 2008] is a language with functional and logical properties. The syntax is inspired by Lisp. So all expressions follow a prefix notation and are list based, like `(name p1 p2 p3 ...)`. Similar to SWRL, variables are names prefixed with a question mark. The language allows the definition of rules based on predicates, as well as the definition of new functions and predicates. Therefore the Jess language is not limited to the predicates induced by OWL properties and classes as SWRL is.

3 Model of the Knowledge Base

The main goal of my thesis is the modeling of knowledge of middleware-oriented architectural styles. Current approaches of defining architectural styles focus on component and connector types and their interaction. These modeling approaches use architecture description languages (ADL) like ACME. ADLs allow to express the composition of architectures and sometimes the composition of architecture styles, but lack the ability to express properties of styles as a whole. For instance they cannot answer the question: Is a style X a good choice for the migration from an architecture based on style Y? To answer this question the meta-information about architectural styles is needed, especially for middleware-oriented styles.

For example, a system based on Cocoon [Cocoon 2008] and Avalon [Avalon 2007] shall be migrated to a new system where the deprecated Avalon is replaced by another framework. Therefore information about the compatibility of middleware-oriented styles is necessary. However the goal of this thesis is not to replace ADLs or architectural style ontologies (see [Pahl et al. 2007]), the goal is to augment ADLs and architectural style ontologies with a knowledge data base.

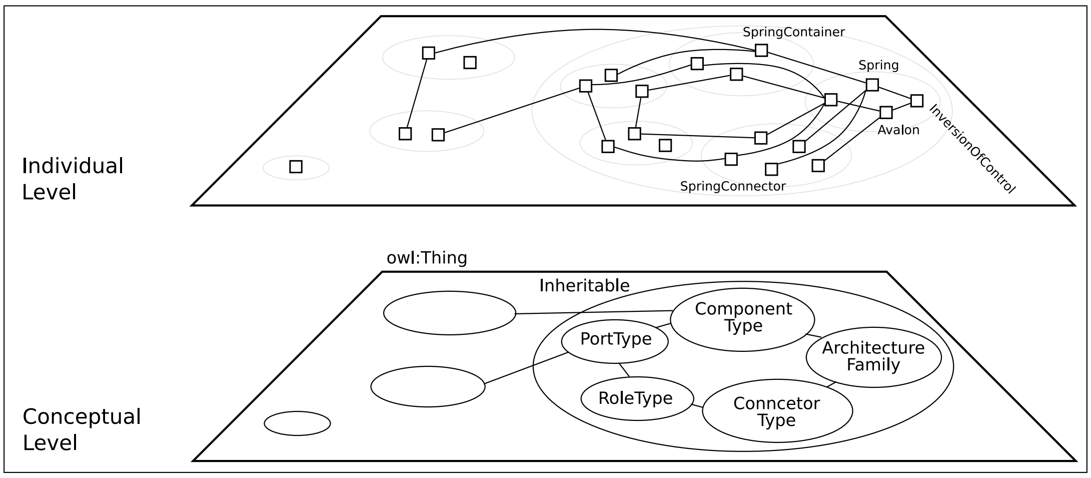


Figure 3.1: Illustration of the individual and conceptual levels in a knowledge base and their relation to each other.

In this chapter the domain model of the knowledge base is explained. A *domain model* is a conceptual system describing entities or terms and their basic relationships within a defined field of interest. The field of interest of this work are middleware-oriented architectural styles, and the entities of the model are styles, middleware-platforms, and related objects.

A problematic aspect in this chapter is related to the term property. It is used in the context of OWL for object or datatype properties, for properties of architectural styles, and on the conceptual level for class properties, mostly formulated as property restrictions. It is important to distinguish between these property types, because they have different semantic meaning. For example a class can have a parent class. This relationship is represented by a class property. However individuals can have an object property for a relationship between two individuals with the same meaning.

The figure 3.1 on page 37 shows two distinct levels of the knowledge base. The lower layer represents concepts and the relations between them, symbolized through lines. These concepts are represented by OWL classes and the relations are defined through object properties and restrictions to these object properties. The inheritance of classes is represented by the circles in the lower layer. A class which is inside another class is considered a sub-class. The upper layer represents the individuals of the knowledge base and their concrete relationships to each other. The gray circles in the upper layer represent the classes from the conceptual level and symbolize the relationship of individuals to classes.

The construction of a knowledge base is a difficult task, and many different aspects have to be kept in mind. The bare definition of concepts do not suffice for a basic knowledge base. First, the domain of the knowledge base is described and from the description the requirements are deduced for the domain model.

In section 3.1 the domain model is explained and how it is used to support software engineers in their decision making process. Section 3.2 discusses two possible basic approaches for the domain model based on an simplified example domain. Section 3.3 introduces the founding class and property structure of the ontology, which represents the most basic relationships in the ontology. In a model meta-model approach, this would represent the meta-model part. However it can also be compared to libraries in programs where the founding classes and properties are a basic library, like `libc` on Unix systems. Section 3.4 introduces some guidelines for populating the knowledge base, which are utilized in fifth section. Section 3.5 introduces a basic set of styles and related individuals as well as a classification and guidelines for properties and individuals for middleware-oriented architecture styles, which should be helpful in the further knowledge base development.

3.1 Requirements for the Knowledge Base

The domain for this knowledge base is induced through the usage of middleware-oriented architectural styles in the MidArch method. Therefore the entities are derived from that domain. A middleware-oriented architectural style can be defined through component, connector, port, and role types, and additional configurations. The ontology-based approach for architectural styles [Pahl et al. 2007] is a good basis for my ontology, because it defines basic classes, properties, and individuals on the basis of a component and connector model.

Before I can deduce requirements for the knowledge base and its domain model, I have to describe usage scenarios for the knowledge base. These scenarios are determined by the MidArch method and particular by the tasks *Specify New MidArch Style* and *Select MidArch Style Candidate* (see section 2.4).

In task 3A (see figure 2.4), new styles are defined or existing ones are refined. During this definition and refinement new meta-information about the styles can be found and codified. In task 3B, a software engineer selects one or more styles as a basis for candidate architectures. This selection process can be based on the GQM method [Basili et al. 1994], however a knowledge based approach can be helpful as well and improve the decision making process.

These two use cases induce directly two requirements. First, there has to be a way to store meta-information in the knowledge base, and second, there must be a retrieval method. These requirements can be broken down in more distinct requirements.

3.1.1 Storing Meta-Information

The storing meta-information requirement has four aspects induced by the domain model. First, basic concepts, like style, component and connector type must be defined. Second, the relationships between these concepts must be defined through a set of properties. Third, additional properties for general knowledge about the style must be defined as well. And fourth, ranges for these properties have to be defined to ensure a basic vocabulary, which introduces enough cohesion for the knowledge base. Cohesion means in this case, that the nodes (i.e. individuals) in the knowledge base are interconnected and no group exists, which is separate from others.

The foundation of the knowledge base is defined through basic concepts. While the knowledge base is an open system which allows and encourages later extension, it is important to define a common basis. This basis is build on a set of predefined classes induced by the component and connector model of architectural styles.

The classes alone however, do not suffice to describe the foundation of the knowledge base. It is important to define relationships between these concepts. These relationships are expressed through object properties. One type of properties express the cognation of individuals. For instance that a style is the parent style of another. Other properties must describe facts like, styles have component and connector types. And finally there have to be properties to describe meta-information of styles and their compositional elements. In fact these properties are relevant to support the engineer in the decision making process.

The properties used in meta-information modeling, introduce in most cases new sets of values, which have to be defined through classes and individuals. This also applies to the initial set of classes and the their basic population with individuals.

3.1.2 Retrieve Meta-Information

The second requirement is about knowledge retrieval and can also be divided into several sub requirements. First, if the classes and the object properties are defined in a useful way information can be automatically inferred which can lead to new knowledge. Most importantly the graph implied by individuals and their relationships must be cohesive, for inference algorithm to produce more knowledge. This allows the user to browse the knowledge base to find information about styles and style aspects.

Beside browsing, the process of querying or questioning the knowledge base is another way to gain information. The intriguing feature of queries and questions is, that they allow reasoning with the stored knowledge based on a defined problem.

For example the technology Avalon [Avalon 2007] is deprecated and therefore systems using this framework should migrate to Spring [Spring 2008] or another *Inversion of Control* technology. For Cocoon-based systems this is recommended, because Cocoon switched from Avalon to Spring as its foundation technology with its new release [Cocoon 2008]. Such knowledge could be retrieved by studying the Cocoon, Avalon, and Spring web-pages, however the knowledge base could answer the question as well, when initialized properly.

3.2 Ontology modeling Approaches

The basic elements of ontologies in OWL are classes, individuals and properties. However, the use of these elements to design an ontology depends, on the perspective of the designer and how classes and individuals are seen.

This thesis' ontology is about architectural styles, component and connector types, and many other concepts. These concepts could be coded either as classes to represent sets of elements or as individuals. Both approaches have their advantages and disadvantages, which are discussed later in this section.

First, I illustrate the issue, with a simple style example representing the *Avalon* and *Spring* middlewares as well as *Inversion of Control* as a general purpose style. Second, I discuss two possible modeling approaches. And finally I discuss their advantages and disadvantages.

3.2.1 Example Knowledge Base

The Avalon framework [Avalon 2007] is mainly used to model web services and is maintained by the Apache Foundation. It is based on the Inversion of Control pattern [Fowler 2004], where components are embedded in a container, which controls these embedded components [Bornhold 2006]. The Spring framework [Spring 2008] is another implementation of the Inversion of Control pattern and has the same main application, which means it is used mainly in web services. However there are differences between both frameworks.

For instance the way they realize the common Inversion of Control pattern. While Spring uses *dependency injection* to vary the connections between components, Avalon

Property	**Avalon**	**Spring**
Inherit concept	Inversion of Control	
Inversion of Control principle	Service Locator	Dependency Injection
Configuration management	Fortress	Java-Beans
Supports pattern	Aspect-oriented Programming	
Supports pattern	Separation of Concern	
Supports pattern	—	Model View Controller
Has documentation	missing	excellent

Table 3.1: Properties of the Inversion of Control based styles Avalon and Spring

uses a *service locator* [Fowler 2004]. Other properties for this example are the way how configuration management is handled in both approaches and which design patterns are supported by both technologies.

Table 3.1 lists some properties in an example set. The exact meaning of the values is here not relevant, however it can been seen that the two styles have common properties, yet the values of these properties can be different. For example the Spring style supports the *Model View Controller* pattern, while this information is not available about Avalon. However both styles support *Separation of Concern* and *Aspect-oriented Programming.* They also have a configuration management however it is implemented with different methods. Such knowledge can be helpful in the assessment of alternatives, because in a migration process the configuration have to be ported as well.

3.2.2 Individual based Approach

The individual approach uses only a minimal set of classes to describe basic concepts and understands architectural styles and the elements belonging to a style as individuals. Each individual has a set of properties. One common property for all three architectural styles is a control container. The OWL DL method of assigning a component type to an architectural style is done by adding `value(`*hasComponentType ControlContainer*`)`.

Without assistance of a rule language, this definition must be added to all three architectural styles. In more complex scenarios with many styles and where styles may even have multiple parent styles, this results in copying many definitions, because properties of the parent style are not inherited by the derived style automatically. This is due to the fact that individuals do not share properties automatically, because the property of inheritance has no syntactic meaning for the reasoner. However this drawback can be attenuated with SWRL rules and a reasoner with SWRL support. The inheritance aspect of *hasComponentType* can be expressed with a SWRL rule. Let's assume the property for inheritance is called *hasParent*, then the rule

$$hasComponentType(?x, ?y) \wedge hasParent(?x, ?z) \rightarrow hasComponentType(?z, ?y)$$

would define that *hasComponentType* is true for style ?z, if there is a style ?x which is parent of ?z and which has the property *hasComponentType* associated with the requested component type.

```
Class(Style partial
   restriction(hasComponentType someValuesFrom(ComponentType))
   restriction(hasParent maxCardinality (1))
)

Class(ComponentType partial owl:Thing)

EnumeratedClass(Pattern
   ModelViewController
   AspectOrientedProgramming
   SeparationOfConcern
)

ObjectProperty(hasComponentType domain(Style) range(ComponentType))

ObjectProperty(hasParent domain(owl:Thing) range(owl:Thing))
ObjectProperty(hasInversionOfControlPrinciple domain(Style) range(
   ServiceLocator
   DependencyInjection
)
ObjectProperty(supportsPattern domain(Style) range(Pattern))
ObjectProperty(hasDocumentation domain(Style) range(oneOf(
      "excellent"
      "good"
      "satisfactory"
      "adequate"
      "inadequate"
      "missing"
   )))

Individual(InversionOfControl type Style
   value(hasComponentType ControlContainer)
)
Individual(Spring type Style
   value(hasParent InversionOfControl)
   value(hasComponentType SpringControlContainer)
   value(hasInversionOfControlPrinciple DependencyInjection)
   value(supportsPattern(AspectOrientedProgramming))
   value(supportsPattern(SeparationOfConcern))
   value(hasDocumentation(excellent))
)
Individual(Avalon type Style
   value(hasParent InversionOfControl)
   value(hasComponentType AvalonControlContainer)
   value(hasInversionOfControlPrinciple ServiceLocator)
   value(supportsPattern(AspectOrientedProgramming))
47     value(supportsPattern(SeparationOfConcern))
48     value(hasDocumentation(missing))
49  )
50  Individual(ControlContainer type ComponentType)
51  Individual(SpringControlContainer type ComponentType
52     value(hasParent ControlContainer)
53  )
54  Individual(AvalonControlContainer type ComponentType
55     value(hasParent ControlContainer)
56  )
57
58  Implies(
59     Antecedent(hasComponentType(?x,?y) hasParent(?z,?x))
60     Consequent(hasComponentType(?z,?y))
61  )
```

Listing 3.1: The Spring and Avalon example coded in an individual-based approach using OWL and SWRL in abstract syntax notation

Listing 3.1 shows the example knowledge base in form of an individual approach. The basic concepts style and component type are defined as classes. The list of known pattern are described by an enumerated class (see section 2.2.2), which is used as range in *supportsPattern* property. The property *hasComponentType* is defined as object property in line 14. The additional rule, which makes this property inheritable can be found at the end the listing (row number 59).

3.2.3 Classes based Approach

The second approach is based on modeling Avalon, Spring and Inversion of Control as classes similar to the approach in [Pahl et al. 2007]. Listing 3.2 illustrates this class based approach.

```
Class(Style partial
   restriction(hasComponentType someValuesFrom(ComponentType))
   restriction(hasParent maxCardinality (1))
)

Class(ComponentType partial owl:Thing)

EnumeratedClass(Pattern
   ModelViewController
   AspectOrientedProgramming
   SeparationOfConcern
)

ObjectProperty(hasInversionOfControlPrinciple
   domain(InversionOfControl)
   range(
      ServiceLocator
      DependencyInjection
))
ObjectProperty(supportsPattern domain(Style) range(Pattern))
ObjectProperty(hasDocumentation domain(Style) range(oneOf(
      "excellent"
      "good"
      "satisfactory"
      "adequate"
      "inadequate"
      "missing"
   )))

SubClassOf(
   Class(InversionOfControl partial
      restriction(hasComponentType someValuesFrom(ControlContainer))
      restriction(hasInversionOfControlPrinciple minCardinality(1)))
   Style
)

SubClassOf(Class(SpringControlContainer partial  ... )
   ControlContainer)
SubClassOf(Class(AvalonControlContainer partial  ... )
   ControlContainer)

SubClassOf(Class(Spring partial
      restriction(hasComponentType
         someValuesFrom(SpringControlContainer))
      restriction(hasInversionOfControlPrinciple
         value(DependencyInjection))
      restriction(supportsPattern value(AspectOrientedProgramming))
      restriction(supportsPattern value(SeparationOfConcern))
      restriction(hasDocumentation value(excellent)))
   InversionOfControl
)
SubClassOf(Class(Avalon partial
      restriction(hasComponentType
         someValuesFrom(AvalonControlContainer))
      restriction(hasInversionOfControlPrinciple
         value(ServiceLocator))
      restriction(supportsPattern value(AspectOrientedProgramming))
      restriction(supportsPattern value(SeparationOfConcern))
      restriction(hasDocumentation value(missing)))
   InversionOfControl
)
```

Listing 3.2: The Spring and Avalon example coded in an class-based approach using OWL in abstract syntax notation

The foundation for this knowledge base is identical to the individual based approach. The concepts style and component type are defined as classes. However the distinct styles are defined as specialization of these basic classes. This means all styles are defined as sub-classes of the class *Style*. The styles *Avalon* and *Spring* are sub-classes of *InversionOfControl* which is itself a sub-class of *Style*. This has the effect that all definitions which apply to *Style* also apply to *Spring*.

Style properties in this approach are defined as object properties and expressed as restrictions in the class specifications. A restriction allows to restrict the value of a property to a certain set of values (see section 2.2.2). Because the inheritance is realized through sub-classing, restriction of the parent class are also valid in the descendant class. This results in implicit property inheritance, which means the inheritance of properties are not expressed through explicit rules, but through the inheritance of model classes.

3.2.4 Discussion

The class based approach has several advantages over the individual based approach. First architectural-styles can be defined as sub-classes of other styles, just like in [Pahl et al. 2007]. This is even more useful when looking at the component types. While the individual approach assigns new component types to Spring and Avalon called *Spring-ControlContainer* and *AvalonControlContainer*, the class approach just restricts the use of component types to *SpringControlContainer* and *AvalonControlContainer* respectively. This leads in the individual approach to a situation where *ControlContainer* and a specialization of this component type are associated with a style. In the class-based approach the property is just restricted a little more to allow only the specialized version. Through its implicit inheritance property the class-based approach does not need explicit rules to simulate inheritance.

However the class-based approach has also some disadvantages. The rule languages are all based description logic for decidability reasons. These languages interpret classes as unary and properties as binary properties, and the universe of values is formed by the individuals of the knowledge base (see section 2.2.3). To query such knowledge base, a rule language can be used as well, because a query is just a rule with an empty consequent (see section 2.2.3). Therefore it is not possible to use rule based querying in the class based approach. While OWL Full would allow the use of classes as individuals, the logic OWL Full implies is undecidable. To be more precise the logic, which is necessary to implement OWL Full, allows ontology structures which are no longer decidable. This is a real problem for automatic reasoning. Therefore OWL DL is the only suitable language and therefore rule languages cannot be used to query the knowledge base.

The existing reasoner and rule engines are designed to work with OWL DL. However some of them are able to process OWL DL together with a rule language like SWRL (see section 2.6.2 and 2.6.3). Through these rule languages the expressiveness of OWL DL can be enriched and explicit inheritance can be implemented. Although this leads to a lot of definitions, it allows to work explicitly with inheritance, which gives a more direct control over what can be inherited and what is better left out. The implicit inheritance can be compared to an *opt-out* policy, where it has to be stated explicit that something shall not apply, while the explicit inheritance can be compared to an *opt-in* policy, where it has to be stated explicit that something shall apply.

While the class based approach has a lot of advantages, its limitation in querying is rather unpleasant. The knowledge could be modeled in a cleaner and more consistent way, but querying is not possible on such a structure. consequently the individual approach has to be applied. Furthermore future rule languages such as [Boley and Kifer 2007] have negation operators, which allow the definition of rules, which block inheritance of a more generic component type when a specialized one is defined. So this disadvantage is merely a temporary one.

3.3 Basic Ontology

In the previous section I discussed two possible ways to construct the knowledge-base. Both have their advantages and disadvantages. However only the individual approach has all features needed to develop a working knowledge base.

Before I can start working with the knowledge base, I have to define its basic concepts and relationships between these concepts. Because knowledge bases tend to grow, it is possible to add further concepts later. This is necessary to express new ideas supported by the documented styles.

The basic concepts for the knowledge base are architectural styles, component, connector, role and port types, as explained in [Pahl et al. 2007]. Relationships between these concepts are expressed through properties. In the following two subsections the elementary concepts are introduced, followed by the object and datatype properties. As a definition language OWL DL is used in abstract syntax notation.

3.3.1 Basic Concepts

The basic concepts are derived from [Pahl et al. 2007] enriched with the concepts pattern and technology. The central concept is architectural style, however the knowledge base does not store style descriptions itself, its is merely an addition to a style database. Therefore the term *architecture family* is used. Part of the family description are information about the component, connector, port and role types. These parts are represented by concepts as well. Outside the scope of [Pahl et al. 2007] two additional concepts are introduced to the basic set of classes. The first are architecture-level design patterns, like *Separation of Concern* [Dijkstra 1982] and the second is technology. In this case I use technology to subsume software libraries and packages which are designed to address a specialized technical problem, such as abstraction of database access or in implementing certain design patterns. Technologies are relevant because the middleware-oriented styles of the MidArch method are based on technology or at least closely related to certain technologies.

The five classes based on concepts from [Pahl et al. 2007] have all the capability to inherit properties from other individuals. Therefore the class *Inheritable* in listing 3.3 models this capability. The inheritance is expressed by the *hasParent* property.

An architecture family describes a set of architectures having certain similar characteristics. Based on the concepts from [Pahl et al. 2007], architecture families comprises

```
Class(Inheritable partial
   restriction(hasParent)
)
```

Listing 3.3: Primary class for inheritable individuals

```
SubClassOf(Class(ArchitectureFamily partial
      restriction(hasComponentType someValuesFrom(ComponentType))
      restriction(hasConnectorType someValuesFrom(ConnectorType))
      restriction(hasParent someValuesFrom(ArchitectureFamily)))
   Inheritable
)
SubClassOf(Class(MiddlewareOrientedArchitectureFamily partial)
   intersectionOf(ArchitectureFamily Technology)
)
```

Listing 3.4: Definition of the architecture family concept

of component and connector types. The definition in listing 3.4 expresses the description of *ArchitectureFamily* through two restrictions, which demand that individuals, this means concrete architecture families, have component and connector types from the class *ComponentType* and *ConnectorType* respectively. Also *ArchitectureFamily* inherits the restrictions on the *hasParent* property from *Inheritable*. However *Inheritable* does not restrict the use of parent concepts to architecture families, therefor this is done with an additional restriction, which limits the values to *ArchitectureFamily* individuals.

The second class defined in listing 3.4 is a specialization of architecture family. The specialization is called *MiddlewareOrientedArchitectureFamily* and represents technology-based and therefore middleware-oriented architecture families. Therefore a middleware-oriented architecture family has two parent concepts, which are expressed through the `intersectionOf` statement.

```
SubClassOf(Class(ComponentType partial
      restriction(hasPortType someValuesFrom(PortType))
      restriction(hasPortType minCardinality(1))
      restriction(hasParent someValuesFrom(ComponentType)))
   Inheritable
)
```

Listing 3.5: Definition of the component type concept

The building blocks of architectures are components. Components follow a common structure. They are normally described by ports and names. Like architectures, components can be grouped by common properties. These groups form component types. The class *ComponentType* defined in listing 3.5 describes the concept of component types. These types have common port types, which are assigned with the property *hasPortType*.

In addition components need at least one port and therefore component types need at least one port type, which expressed by the second restriction in listing 3.5. Component types can be specializations of other component types, just like architecture families could be specializations of other families. Therefor the definition of *ComponentType* is based on *Inheritable*. Analogue to the definition of *ArchitectureFamily*, the *hasParent* is restricted to *ComponentType*.

```
SubClassOf(Class(ConnectorType partial
      restriction(hasRoleType someValuesFrom(RoleType))
      restriction(hasRoleType minCardinality(2))
      restriction(hasParent someValuesFrom(ConnectorType)))
   Inheritable
)
```

Listing 3.6: Definition of the connector type concept

Beside components, connectors are the second description element to compose architectures. Like components, connectors can be grouped to connector types. The class *ConnectorType* (see listing 3.6) reflects these types. Where components have ports, connectors have roles. These role types are referenced by the *hasRoleType* property. The cardinality of role types are a minimum of two in order to express that a connector connects at least two components. A connector with only one end would not make any sense at all, since it would not connect anything.

```
SubClassOf(Class(PortType partial
      restriction(acceptsRole someValuesFrom(RoleType))
      restriction(acceptsRole minCardinality(1))
      restriction(hasParent someValuesFrom(PortType)))
   Inheritable
)
SubClassOf(Class(RoleType partial {
      restriction(connectsTo someValuesFrom(PortType))
      restriction(connectsTo minCardinality(1))
      restriction(hasParent someValuesFrom(RoleType)))
   Inheritable
)
```

Listing 3.7: Definition of the port and role type concepts

Ports describe the way how components can communicate with their environment. On the abstract component type level, these ports are represented by port types defined in listing 3.7. Port types can be specializations of more general port types, therefore they need to inherit some properties from these parent types. The inheritance is realized through sub-classing *PortType* from *Inheritable*. Similar to component and connector types, the inheritance property has to be restricted to port types only.

While components have ports, connectors have roles. And therefore there are role types for connector types. In listing 3.7 both classes are defined. While *PortType* requires that individuals have to accept at least one role type, *RoleType* individuals have to connect to at least one port type. Like all other sub-classes of *Inheritable* they have an additional restriction on the type of parent individuals.

```
Class(DesignPattern partial owl:Thing)
```

Listing 3.8: Definition of the architecture-level design pattern concept

The class *DesignPattern* represents architecture-level design patterns like *Separation of Concern* or *Model View Controller*. They do not have any sub-structure in the basic ontology, but they are required to describe certain characteristics of technologies and middleware-oriented architecture families. Therefore the concept *DesignPattern* is represented by a class without any restrictions (see listing 3.8). However it can be necessary to extend the definition in future to address properties and relationships of design pattern. This can especially be useful when this ontology is connected to a design pattern database.

```
Class(Technology partial
   restriction(supportsPattern someValuesFrom(DesignPattern))
)
```

Listing 3.9: Definition of the technology concept

Associated with design pattern is the class *Technology* which represents certain technologies. This can be programming languages, frameworks which are not represented by architecture families, software packages, or libraries. Additionally this *Technology* class allows a more informal view to technologies than architecture families. Because *MiddlewareOrientedArchitectureFamily* is an intersection of *ArchitectureFamily* and *Technology*, a technology definition can be transformed in a more formal description without breaking references.

3.3.2 Basic Properties

In the previous section the foundational classes have been introduced. Along with these classes some properties have been used. In this section these properties are explained and defined, along with a additional properties and rules in order to implement basic relationship functionality into the ontology. Because this is not always possible using OWL alone, SWRL rules are used as well.

The *hasParent* property, defined in listing 3.10 represents a relation between individuals of the class *Inheritable*. Restrictions to this property are specified in the sub-classes of *Inheritable*. The property *hasRelative* is completely coded in SWRL and allows a broader view on relationships of individuals.

```
ObjectProperty(hasParent domain(Inheritable) range(Inheritable))

Implies(
   Antecedent(hasParent(?x,?y))
   Consequent(hasRelative(?x,?y))
)
Implies(
   Antecedent(hasRelative(?x,?y))
   Consequent(hasRelative(?y,?x))
)
Implies(
   Antecedent(hasRelative(?x,?y) hasRelative(?y,?z))
   Consequent(hasRelative(?x,?z))
)
```

Listing 3.10: Definition of the *hasParent* and *hasRelative* properties

```
Implies(
   Antecedent(hasParent(?x,?y))
   Consequent(hasAncestor(?x,?y))
)
Implies(
   Antecedent(hasAncestor(?x,?y) hasAncestor(?y,?z))
   Consequent(hasAncestor(?x,?z))
)
Implies(
   Antecedent(hasParent(?x,?y) hasParent(?z,?y))
   Consequent(hasSibling(?x,?z))
)
Implies(
   Antecedent(hasParent(?x,?y))
   Consequent(hasChild(?y,?x))
)
Implies(
   Antecedent(hasChild(?x,?y))
   Consequent(hasDescendant(?x,?y))
)
Implies(
   Antecedent(hasDescendant(?x,?y) hasDescendant(?y,?z))
   Consequent(hasDescendant(?x,?z))
)
```

Listing 3.11: Additional rules to complement basic properties

Additional rules (see listing 3.11) are defined to complement the basic set of relationships between individuals and to ease later property development and ultimately the definition of queries. First, a more general form of *hasParent* property, the *hasAncestor* property is defined. The *hasAncestor* property allows tracing the complete inheritance path. The second rule allows finding siblings of an individual. These individual relationships can be useful when searching for similar inheritables.

The rule *hasChild* implements the inverse of *hasParent*. This is a convenience rule, because *hasParent* would suffice. The last rule *hasDescendant* make up these complementary rules. It allows finding all descendants of an individual, which can be helpful when searching for more specialized version of an inheritable individual.

```
ObjectProperty(hasComponentType
   domain(ArchitectureFamily) range(ComponentType))
Implies(
   Antecedent(hasComponentType(?x,?y) hasParent(?z,?x))
   Consequent(hasComponentType(?z,?y))
)

ObjectProperty(hasConnectorType
   domain(ArchitectureFamily) range(ConnectorType))
Implies(
   Antecedent(hasConnectorType(?x,?y) hasParent(?z,?x))
   Consequent(hasConnectorType(?z,?y))
)
```

Listing 3.12: Definition of the *hasParent* property

The properties *hasComponentType* and *hasConnectorType* build the relationships between architecture families and component or connector types. Since component and connector types are inherited, both properties are complemented by SWRL rules to realize their inheritance. The definition in listing 3.12 utilizes therefore the previously defined property *hasParent*.

```
ObjectProperty(hasPortType domain(ComponentType) range(PortType))
ObjectProperty(hasRoleType domain(ConnectorType) range(RoleType))
ObjectProperty(isPortTypeOf inverseOf(hasPortType))
ObjectProperty(isRoleTypeOf inverseOf(hasRoleType))
```

Listing 3.13: Definition of the *hasParent* property

In listing 3.13 the properties *hasPortType* and *hasRoleType* are defined. They define the relations between component types and port types and between connector types and port types respectively. The two properties *isPortTypeOf* and *isRoleTypeOf* are inverse properties of *hasPortType* and *hasRoleType*. These definitions allow a rule engine or an OWL reasoner to use the declaration of component and port type mappings as well as connector and port types in both directions. It also allows the user to define the relation from different points of views.

While ports belong to certain components and roles to certain connectors, there exists also a relationship between ports and roles. Therefore, on an abstract level, between the

```
ObjectProperty(acceptsRole domain(PortType) range(RoleType))
ObjectProperty(connectsTo inverseOf(acceptsRole))
```

Listing 3.14: Definition of the *hasParent* property

port and role types a relation has to exist. To express this relation, two properties are defined. In listing 3.14 the property *acceptsRole* is defined as relation between *PortType* and *RoleType*. The other property *connectsTo* is defined as the inverse of *acceptsRole*, because every time a relationship between a port and a role type is defined the inverse of it is also true.

```
ObjectProperty(supportsPattern
     domain(Technology)
     range(DesignPattern)
)
```

Listing 3.15: Definition of the *hasParent* property

The last property of the basic ontology, defined in listing 3.15, addresses the reality that certain technologies support diverse design patterns. Thus middleware-oriented architecture families can also support certain design pattern, because they are technologies and architecture families as defined in listing 3.4.

3.4 Modeling Issues and Solutions

Before populating the knowledge base with the first individuals, simple modeling guidelines have to be introduced to prevent a chaotic approach. This has to be done in two steps. In a first step typical problems are characterized and in a second step guidelines are expressed to avoid the introduced problems.

3.4.1 Common Knowledge Base Modeling Problems

The definition of knowledge can be cumbersome, because it is often not clear how to express a certain aspect of knowledge. It can be difficult to decide when to use classes, properties, rules, or combinations of them.

The first common problem is using enumerated and general classes. An enumeration allows to define a fixed set of individuals for a class. The class is then completely defined through this enumeration. In some cases this is enough, but when these individuals have to be extended with properties and these properties shall apply to restrictions then such enumeration is impractical, because enumerated classes do not have property restrictions on their own.

Similar to the class problem, the use of object or datatype properties presents us with a second problem. While integer based datatype properties imply an ordering, which is normally not available for object properties, the meaning of the values can change in various contexts.

For example grades are expressed in 1 (excellent) to 6 (unsatisfactory) in Germany. However in other countries for instance the Netherlands, 10 means (excellent) and every thing lower than 5 is inadequate. Furthermore in some countries numerical values are not used to grades at all. Therefor it can be wise to use a named approach, which will could lead to an individual based representation of grades.

The last common problem is about when and what to express in rules or properties. When building additional rules in OWL properties with symmetric, transitive, or functional characteristic, then these rules might redefine some of those characteristics. Therefor it could be concluded that it might be easier to define all characteristics of the OWL property with rules. However the decision in using OWL or SWRL to describe these characteristics is influenced by many circumstances. Some tools and software packages have problems to transfer the OWL characteristics to the rule reasoning domain. Also it can be easies for a knowledge designer to keep all rules about classes and properties in one view. If some of the rules are implicitly defined through OWL statements and others are defined in SWRL, it can be difficult to keep them all in mind during the design process.

3.4.2 Knowledge Base Modeling Guidelines

The previous section described three typical problems while defining classes, properties, and individuals for the knowledge base. In this section guidelines which help to solve these problems are explained and remarks are made.

The general problem about the right usage of classes, properties, and individuals can be solved by asking a set of questions. First, is the knowledge to be described a relationship or at least includes a relationship or an entity? If it is a relationship then this should be described with a property. If not it must be a class or an individual.

Second, is the entity a concept or is it supposed to appear in a query result? This question is most difficult, because most entities are somehow a concept and an individual. For example, a cat is an individual type of mammal, however it could also be seen as a class, because there are sub-classes of cats and in the end there are also individual lifeforms called i.e. Kitty. The solution to this problem lies in the answer to the second part of the question: Is it supposed to appear in a query result? If the answer is yes then the entity must be modeled as individual. Entities, which are possible results of a query must be individuals, because rule engines, such as Jess (see section 2.6.4), can only return individuals and not classes in a query result.

All individuals must belong to a class and it is best not to use *owl:Thing* as class, because it is too generic. Rather a suitable class for the new individual must be found. However, sometimes there is no suitable class, then a new class must be defined. When defining a new class, the class should be on the one hand generic enough to hold more than one single individual, and on the other hand it should not be too arbitrary. A good class also applies one or more restrictions to individuals belonging to it. If it is possible to inherit features from other classes then the new class should be a sub-class of these classes.

Classes in OWL can be defined through restriction and with various set operators from other classes. Another way to define them is by enumerating their individuals. The question is: When to do use enumerated classes? This question can definitely be answered, if a class has a definite list of individuals which most likely do not change in the future and the properties of these individuals do not matter then a enumerated class can be used. However in many cases it cannot be known if these requirements are met. Therefore, if in doubt it is a good advice to use classes with restrictions and define the individuals separately, yet enumeration must not be dismissed completely.

For example if grade values have to be defined then enumerations can be used. If the property can be expressed by a datatype property, but the datatype property is not verbose enough for the software engineer, then an enumerated class is a suitable for defining the values for that property.

This leads directly to the problem of datatype or object properties. Datatype properties are useful when calculations have to be performed. For example most properties induced by metrics are easily coded in datatype properties. If the values are more descriptive, like in the above example about the grades (excellent to unsatisfactory), then an enumerated list of values is a better solution. And therefore object properties are the right property type.

The last method to describe knowledge are rules. Rules are very powerful and can be used for many purposes. They are able to express transitive and symmetric relational characteristics of OWL properties, as well as knowledge which can be described by a set of relations. For example, the rule expressing the sibling relation of inheritable individuals, is based on arranging the *hasParent* property in such a way as to express the definition of siblings. This allows to define a sibling relationship between two individuals without explicit definition. Rules are therefore very useful to express general knowledge. However some relationships can also be defined with object properties, therefore the transitivity and symmetry can be expressed also with OWL properties (see section 2.2.2).

3.5 Initial Application of the modeling Approach

In the previous sections basic concepts and relations were defined for the ontology as well as guidelines for knowledge specification. These concepts and relations are the skeleton of the knowledge base. In this section the knowledge base is populated with its first individuals to implement a common set of basic knowledge so that other engineers can use it as a basis of their definitions. The source for these individuals are the classification in [Shaw and Clements 1997] and the analysis of middleware-oriented styles in [Bornhold 2006].

Further, subsections discuss the classification of styles in [Shaw and Clements 1997] and the Avalon and Cocoon styles from [Bornhold 2006] with the goal to define individuals and properties for the knowledge base. The last subsection addresses the problem of finding suitable properties for architecture families, component and connector types.

3.5.1 Knowledge based on a Field Guide of Boxology

Shaw and Clements [1997] introduced in their paper *A Field Guide to Boxology* a preliminary classification of architectural styles. Their approach uses five distinct categories or dimensions to classify architectural styles. These categories are: (1) the type of components and connectors; the way (2) control and (3) data is shared, allocated, and (4) communicated through the system; (5) the suitable reasoning type which is compatible with the style. Below I explain these five categories and the resulting meta-properties and utilized classes and individuals for this classification.

Component and Connector Types

The basic ontology already defines the concepts component and connector types. In [Shaw and Clements 1997] component and connector types are classified by processes, programs, objects, or transducers. These basic types symbolize component types in the ontology. Corresponding to these elementary component types the paper from [Shaw and Clements 1997] introduces connector types. They define character and data streams, as well as message protocols. The model in [Shaw and Clements 1997] does not include port or role types, therefore these have to be defined in respect to the associations between the component and connector types. Listing 3.16 shows the component and connector types introduced by [Shaw and Clements 1997] as well as the role and port types.

The component types *Process*, *Program*, and *Object* are too generic to assign them port types. These component types are merely there to define basic vocabulary. Therefor other component types, which specify more definite entities, can be distinguished by their relationship to these basic component types. The same applies to the connector type *MessageProtocol*.

Control Flow

A second classification topic in [Shaw and Clements 1997] is control, which has the three aspects topology, synchronicity and binding time. The topology describes the direction of the control flow and the connections between components. For instance a pipeline has a linear topology and the control relationship between the main program and subroutines is hierarchical. Shaw and Clements [1997] define six distinct topologies for the control flow (see listing 3.17).

The aspect of synchronicity describes how the state of dependent components is to the control state of other components (see listing 3.18). In a *synchronous* system, the components synchronize regularly and often. In *asynchronous* systems, components do not interact or synchronize their operations very often. An *opportunistic* system involves no control from one component to another. For example agents work completely independent from each other.

Beside these three synchronicity values, systems can be *parallel* or *sequential*. However the sequential value makes only sense in lockstep systems. All other systems are parallel [Shaw and Clements 1997].

```
Individual(Process type(ComponentType))
Individual(Transducer type(ComponentType)
   value(hasParent Process)
   value(hasPortType InputPort)
   value(hasPortType OutputPort)
)
Individual(Program type(ComponentType)
   value(hasParent Process)
)
Individual(Object type(ComponentType))

Individual(InputPort type(PortType)
   value(acceptsRole(OutputStream))
)
Individual(OutputPort type(PortType)
   value(acceptsRole(InputStream))
)

Individual(Stream type(ConnectorType)
   value(hasRoleType InputStream)
   value(hasRoleType OutputStream)
)
Individual(DataStream type(ConnectorType)
   value(hasParent Stream)
)
Individual(CharaterStream type(ConnectorType)
   value(hasParent Stream)
)
Individual(MessageProtocol type(ConnectorType))

Individual(InputStream type(RoleType))
Individual(OutputStream type(RoleType))
```

Listing 3.16: Component and connector types and their port and role types based on [Shaw and Clements 1997]

The last aspect for control is the binding time, which is the time when the control flow is determined. This can be the time when the program is written (*WriteTime*), compiled (*CompileTime*), invoked (*InvocationTime*), or a run time (*RunTime*). The last option also implies that the control flow may change during run time.

Data Flow

The third classification topic is data flow. Data flow can be described with the four properties: binding time, topology, continuity, and mode.

```
EnumeratedClass(Topology
    acyclic
    arbitrary
    hierarchy
    incompleteGraph
    linear
    star
)
ObjectProperty(hasControlTopology
   domain(ArchitectureFamily) range(Topology))
```

Listing 3.17: Properties and related individuals modeling the control topology [Shaw and Clements 1997]

```
EnumeratedClass(Synchronity
    asynchronous
    opportunistic
    parallel
    sequential
    synchronous
)
ObjectProperty(hasSynchronity
   domain(ArchitectureFamily) range(Synchronity))
EnumeratedClass(BindingTime
    CompileTime
    InvocationTime
    RunTime
    WriteTime
)
ObjectProperty(controlBindingAt
   domain(ArchitectureFamily) range(BindingTime))
```

Listing 3.18: Properties and related individuals modeling the aspects synchronity and binding time of control [Shaw and Clements 1997]

The *binding time* corresponds to the control flow property named binding time. It is the time when the data flow is determined. The definition of the property *dataBindingAt* in listing 3.19 uses therefore the same range as *controlBindingAt*.

The *topology* of the data flow between components can have the same characteristics as those of the control flow. Therefore the definition of data flow topology in listing 3.19 is based on the same range. The topology describes the graph of data flow in an architecture style.

The next property is *continuity*. In [Shaw and Clements 1997] continuity has two aspects. The first aspect describes whether the data flow in continuous, like in a streaming application, or sporadic, like in a chat program. The second aspect is the data intensity. Data intensity can be high, which means the application relies more on data transmission

```
ObjectProperty(dataBindingAt
   domain(ArchitectureFamily) range(BindingTime))

ObjectProperty(hasDataTopology
   domain(ArchitectureFamily) range(Topology))
```

Listing 3.19: Properties for binding time and topology of data flow

than on CPU power, or low, which indicates that the application is more compute intensive. Because these two aspects can be modelled as two separate properties, I decided to define them as separate properties (see listing 3.20).

```
EnumeratedClass(Continuity
   continuous
   sporadic
)
ObjectProperty(hasContinuity
   domain(ArchitectureFamily) range(Continuity))
EnumeratedClass(DataIntensity
   low
   high
)
ObjectProperty(hasDataIntensity
   domain(ArchitectureFamily) range(DataIntensity))
```

Listing 3.20: Properties for data flow continuity and intensity

The *mode* is the last property of data flow to describe. The mode describes the way how data is made available throughout the system. The *broadcast* mode describes styles where data is transmitted to all components and only the receiving components decide whether they do something with the data or ignore it. A more sophisticated variant of broadcast is *multicast*, because the sender defines a set of receivers. This minimizes traffic. In *shared* mode components make data available in a shared place, where every component has access.

For example database systems or systems which support shared memory areas can be seen as such systems. A close relative to shared mode is *copy-in-copy-out* mode, where components take data from the public store, modify it, and re-insert it into the public store. The last mode to describe is the *passed* mode. In an object style, data is *passed* from component to component, which means a data object leaves one component and is then only available to the other component. The definition of modes and the corresponding property are shown in listing 3.21.

```
EnumeratedClass(Mode
    broadcast
    copy-in-copy-out
    multicast
    passed
    shared
)
ObjectProperty(hasMode
    domain(ArchitectureFamily) range(Mode))
```

Listing 3.21: Different modes of data flow and sharing.

Control and Data Interaction

Data and control flow have similar properties, as already explained. While binding time is defined in the same way, the user can define a specialized rule to find out if data and control flow are the same or different. The topology however is more complex. For instance if both topologies are arbitrary, then this does not mean that the topologies have the same edges and nodes. Therefore this has to be specified in an additional property. Also the flow direction of data and control could be in the same or opposite direction. Two properties are defined (see listing 3.22) to cope with this aspect.

```
DataProperty(isIsomorphic
    domain(ArchitectureFamily) range(boolean))
DataProperty(hasSameFlowdirection
    domain(ArchitectureFamily) range(boolean))
```

Listing 3.22: Control and data flow shape and direction

Type of Reasoning

The overall design of an architecture and its underlying style require different analysis approaches [Shaw and Clements 1997], which imply different architecture modeling concepts. For instance a system of parallel working components can be modeled by non-deterministic state machines, while another system is a sequence of atomic steps and therefore can be modeled as a composition of functions [Shaw and Clements 1997]. These different modeling concepts lead to different ways of reasoning about them. However these ways not represented by a property in the knowledge base, because the architectural styles represent this aspect by themselves.

3.5.2 Knowledge based on Middleware-oriented Styles

The middleware-oriented styles introduced in [Bornhold 2006] contain different meta-properties which are not directly expressed through their ADL description. Previously in section 3.2 some properties of an Avalon-style have already been noted. In this subsection they are discussed in more detail and formalized in OWL and SWRL.

The middleware-oriented styles used in [Bornhold 2006] do not necessarily lead to properties which can be used in a wide variety of style descriptions. However they allow to introduce different sources and classes of properties. Beside that, Spring and Avalon are frameworks which belong to the set of Inversion of Control technologies. Other frameworks with a similar target might share features and therefore properties with Spring or Avalon.

The Inversion of Control styles Avalon and Spring, as specified in section 3.2, have different ways for implementing the Inversion of Control pattern. One aspect of this difference is the use of a *ServiceLocator* component in Avalon while in Spring the container is responsible for dependency control. Another aspect is way how dependency injection is realized. The enumerated class *DependencyInjectionTypes* in listing 3.23 comprises these different types.

```
EnumeratedClass(DependencyInjectionType
    InterfaceInjection
    ConstructorInjection
    SetterInjection
)
ObjectProperty(hasDependencyInjectionType
    domain(ArchitectureFamily) range(DependencyInjectionType))
Implies(
    Antecedent(hasAncestor(?x, DependencyInjection)
               DependencyInjectionTypes(?z))
    Consequent(hasDependencyInjectionType(?x,?z))
)
```

Listing 3.23: Different implementation types of the Inversion of Control pattern

The rule included in listing 3.23 requires that individuals, which are descendants of *DependencyInjection*, have also an *DependencyInjectionType*. The type property is an important property for Inversion of Control based architecture families, because the type has significant bearing on cost of implementation and migration. While *ConstructorInjection* and *SetterInjection* allow the use of normal Java objects, called POJO (plain old Java objects), the *InterfaceInjection* styles require components to use certain interfaces defined by the framework.

Another common property of frameworks is testing. Testing allows a developer to check if certain parts of the implementation are working. In listing 3.24 the testing property is described. The class of possible test concepts is declared with two restrictions. These restrictions help to reference a certain test concept with a concrete test technology and a programming language. The restrictions are not defined as required, because not all test concepts are technology or language related.

In listing 3.24 the properties *hasTechnology* and *supportsProgrammingLanguage* are used before explaining them in detail. The definition of both properties is shown in list-

```
Class(TestConcept partial
   restriction(hasTechnology someValuesFrom(Technology))
   restriction(supportsProgrammingLanguage
      someValuesFrom(ProgrammingLanguage))
)
Individual(Stubs type(TestConcept)
   value(supportsProgrammingLanguage Java)
)
Individual(UnitTest type(TestConcept)
   value(hasTechnology JUnit)
)
ObjectProperty(supportsTestMethod
   domain(ArchitectureFamily) range(TestConcept))
```

Listing 3.24: modeling testing abilites in middleware-oriented styles

```
ObjectProperty(hasTechnology domain(unionOf(
    TestConcept))
   range(Technology)
)

ObjectProperty(supportsProgrammingLanguage domain(unionOf(
    TestConcept
    Technology))
   range(ProgrammingLanguage)
)
Class(ProgrammingLanguage partial owl:Thing)
Individual(Java type(ProgrammingLanguage))
Individual(C type(ProgrammingLanguage))
Individual(CPP type(ProgrammingLanguage))
```

Listing 3.25: Supplemental properties and classes used by test ability modeling

ing 3.25. The use of these properties is not limited to the class *TestConcept*, therefore a more open definition is required. Furthermore it is not possible to make a complete definition, because future classes might also use the properties *hasTechnology* and *supportsProgrammingLanguage* and might need to add new technologies and languages.

The property *hasTechnology* is defined as relation between *TestContept* and *Technology*. However *unionOf* is used to express later extensions of the domain. The property *supportsProgrammingLanguage* is defined similar to *hasTechnology*, but with two classes mentioned in the domain. The used class *ProgrammingLanguage* is a stub, as the individuals do not need to have any properties for my purposes. Therefore these definitions are only a declarations of names. A more complex definition of programming languages and their implication on middleware-oriented style selection is not required. However a complete ontology about programming language can be added later even when the names do not match. This is one of the strength of OWL as explained in section 2.2.2.

```
DatatypeProperty(documentationQuality Functional
   domain(ArchitectureFamily) range(oneOf(
      "excellent"
      "good"
      "satisfactory"
      "adequate"
      "inadequate"
      "missing"
)))
```

Listing 3.26: Quality of technology documentation

The aspects about supported patterns and technology are already cared for in the elementary domain model. One last property to be explained in the initial population of knowledge base is for quality of documentation. The example used in section 3.2 is very limited. A more ordinal definition would be more helpful. In listing 3.26 such definition is given. The datatype property is defined as an enumeration of literals with an implicit order where *excellent* is the best and *inadequate* the worst setting. The value *missing* indicates the absence of documentation.

All previous properties focus on the architectural style as a whole. However component and connector types can have properties as well. The example style *InversionOfControl* is not only a named individual, but the style also implies a certain meaning. The most basic information about this style is that components are embedded in other components called containers. The container also controls the embedded components through a connector. This can be expressed by rules and by a property representing that relation.

```
ObjectProperty(controls domain(ComponentType) range(ComponentType))
DatatypeProperty(isContainer
   domain(ComponentType) range(xsd:boolean))
DatatypeProperty(isEmbeddable
   domain(ComponentType) range(xsd:boolean))
```

Listing 3.27: Definition of the embedding and control aspects of the *InversionOfControl* style

Listing 3.27 defines properties to directly express the embedding concept realized by the *InversionOfControl* style. However a more elegant way would be to define the relation of container and embeddable components on a more generic basis.

As long as there is only one container type and one type embeddable component types, there have to be three values set in each architectural style which are descendants of *InversionOfControl*. It would be better, if such property induced by the relationship of component types, can be defined once and the rest is done by the computer. The rule based definition of the *controls* property in listing 3.28 formulates the *controls* relationship based on other properties in one place. In derived styles it is enough to define that

```
Implies(
   Antecedent(
      hasComponentType(?style,?comp) isContainer(?comp,true)
      hasComponentType(?style,?embed) isEmbeddable(?embed,true)
      hasConnectorType(?style,?connector)
      hasPortType(?comp,?portS) acceptsRole(?portS,?roleS)
      hasPortType(?embed,?portC) acceptsRole(?portC,?roleC)
      hasRoleType(?connector,?roleS)
      hasRoleType(?connector,?roleC)
   )
   Consequent(controls(?comp,?embed))
)
DatatypeProperty(isContainer
   domain(ComponentType) range(xsd:boolean))
DatatypeProperty(isEmbeddable
   domain(ComponentType) range(xsd:boolean))
Implies(
   Antecedent(hasAncestor(?x,?y) isContainer(?y,true))
   Consequent(isContainer(?x,true))
)
Implies(
   Antecedent(hasAncestor(?x,?y) isEmbeddable(?y,true))
   Consequent(isEmbeddable(?x,true))
)
```

Listing 3.28: Definition of the embedding and control aspects of the *InversionOfControl* style in a generic approach

a component is a descendant of the container or an embeddable component. Their roles in the style are defined.

The main rule building this connection could be described in English as:

> IF a style has a component type defined as container
> AND the style has another component type defined as component
> AND the style has a connector which connects one port of the container type with one port of the embedded component type
> THEN the container type controls the component type

The conjunction *hasPortType* (?*comp*, ?*portS*) *acceptsRole* (?*portS*, ?*roleS*) *hasPortType* (?*embed*, ?*portC*) *acceptsRole* (?*portC*, ?*roleC*) *hasRoleType* (?*connector*, ?*roleS*) *hasRoleType* (?*connector*, ?*roleC*) from the first rule in listing 3.28 describes the connection between two components with a specific connector. In theory the conjunction could be used to define predicate which holds if the two given components can be connected by the given connector. That would require the definition of a predicate with three parameters. However SWRL does not allow the definition of predicates with more than two parameters. Therefor the conjunction cannot be formulated as a predicate. However it can be used as a template for other rules, which need this expression.

3.5.3 A Classification for Meta-Properties

In the previous section, some properties have been introduced. As source [Shaw and Clements 1997] and [Bornhold 2006] have been used. However the search for properties is difficult and therefore a set of questions and lists with classification for properties can be helpful aids in the property retrieving process. Therefore I am mentioning sources of properties and what kind of questions these properties normally answer for the reader. In a second step, I define a classification for properties on basis of these questions.

A typical source for properties are technical manuals and documentation about frameworks and technologies, which are the basis of middleware-oriented styles. These documentations answer the question how to use a certain framework. Further the decomposition into component and connector types can be made on the basis of such documentation, as already explained in section 2.4. Most manuals mention the application domain for a framework, like workflow management or grid computing.

Another source is user experience from system engineers. Their knowledge is often implicit, which means only they know what they know. However their knowledge is valuable, because it reflects the real use of technologies. So problems and inconsistencies, which are not mentioned in documentation can be extracted from them. Experience answers questions about the usability of technologies in certain scenarios. Additional experience is the source of information about best practice.

The last source for properties is knowledge collected through empirical studies. For example evaluation with the Goal-Question-Metric (GQM) method [Basili et al. 1994] produces knowledge about middleware-oriented styles. The GQM method is used to generate goals, questions for these goals, and a set of metrics to answer the questions. The answers to these question are also a source of knowledge.

These question lead to five areas of knowledge about middleware-oriented styles, which can be seen as a classification scheme for properties. Table 3.2 shows exemplary the previously defined properties in relation to the five areas of knowledge.

The first area comprises the composition of a style. This area is normally targeted by ADLs. For the ontology only a subset of the composition relationships are necessary. These comprise relationships between component, connector, port and role types of a style and the style itself. Additional information about what a thing is, is also part of this area. For example the definition of the term *Container* and its special relationship to other components is of interest. However detailed constraints on these entities are not required and are better formulated in an ADL.

The second area focuses on the purpose of architecture style and their underlying technologies or software frameworks. This includes information about the application domain. For example, small or large enterprises, or system which are mission critical or only supplementary.

The third area is about features of software frameworks and architecture styles based on these frameworks. A feature in this case is any particular property of a style or related styles. For example, the *InversionOfControl* style has the feature of a container and embedded component relationship. And concrete frameworks, like *Avalon* or *Spring*

Area of Knowledge	**Property**
Style composition	*hasParent* *hasComponentType* *hasConnectorType* *hasPortType* *hasRoleType*
Purpose, application domain	x
Features	*hasControlTopology* *hasSynchronity* *controlBindingTimeAt* *hasDataTopology* *dataBindingTimeAt* *hasContinuity* *hasDataIntensity* *hasMode* *isIsomorphic* *hasSameFlowdirection* *hasInversionOfControlType* *supportsTestMethod* *hasTechnology* *supportsProgrammingLanguage*
Quality	*documentationQuality*
Documentation	*documentationQuality*

Table 3.2: The object and datatype properties of the initial knowledge base in relation to the five knowledge areas.

implement different types of Inversion of Control, so the related architecture styles have to reflect these types as properties.

Quality aspects of architecture styles, like complexity, stability, or performance – to name a few – form the fourth area of properties. These quality aspects are often addressed in quality models created (see [Basili et al. 1994]).

The last area is about documentation and support for a framework. Outdated documentation or lack of support for a framework can affect the usability of a framework and therefore influence the selection of an architecture style.

These areas represent a classification for properties. However there might be knowledge which does not fit in these areas, but still is useful information for a software engineer. Therefore the classification shall not be seen as a exclusive system where every property have to fit in, but more as a basis for the search for properties. Furthermore there can be properties, which fit in more than one classification category. This is not a problem, because the purpose of classification is to guide but not to restrain the engineer.

4 Knowledge Modeling Technique

The previous chapter discussed a basic ontology for the knowledge base and guidelines for the development of meta-information of styles. These guidelines are helpful to define properties for styles based on existing information. However they neither integrate the definition task into the corresponding MidArch task 3A (see section 2.4.1) nor explain the full knowledge retrieval process. Therefore this chapter focuses on the complete knowledge retrieval technique which augments the style creation process of the MidArch method [Giesecke 2008].

Before a knowledge retrieval technique is described, the requirements for the technique have to be defined in the context of the MidArch method. Based on these requirements a technique can be created in a three step approach. Section 4.1 discusses the requirements and their context. Section 4.2 motivates a conceptual design of the analysis technique which is complemented by a detailed process description of the method in section 4.3.

4.1 Requirements

The first step towards a knowledge modeling technique is the definition of requirements. These requirements are induced by the MidArch method [Giesecke 2008] which describes processes for gaining information about middleware-oriented styles. This resulting information can be used as input for the modeling technique. The MidArch method is the greater context in which this technique has to fit. This means, it has to follow the process structure and procedures of the MidArch method.

The requirements can be grouped in three categories:

1. Requirements induced by the MidArch style modeling procedure
2. Requirements of different input sources for the knowledge base
3. Requirements related to the view of the software engineer

4.1.1 Style Modeling Procedure Requirements

The style modeling procedure [Giesecke 2008] is briefly explained in section 2.4.1. It is a procedure with five steps. In each step, information is collected or deduced and can be used as input for the knowledge modeling technique. As the information becomes more formal by passing it through the style modeling procedure, the information for the ontology gets more precise. Therefore I can deduct the following requirement:

Requirement 1 *The knowledge modeling technique must follow the style modeling procedure and should define additional jobs for each step in the modeling procedure.*

4.1.2 Input Source Requirements

When looking at the different steps of the modeling procedure, information is gathered from technical documentation and experts, and then compiled into new structured documentation. The structured documentation, the technical documentation and the experts form three distinct input sources for the analysis technique which have to be treated according to their specific properties. This leads to three requirements for information input:

Requirement 2 *The knowledge modeling technique must provide a procedure to convert informal style descriptions from the MidArch style modeling procedure into information for the knowledge base.*

Requirement 3 *The knowledge modeling technique must introduce a method to extract relevant information from long documents for the knowledge base.*

Requirement 4 *The knowledge modeling technique must describe a method to record and transcribe verbal input and transform it into information for the knowledge base.*

4.1.3 User Dependant Requirements

The last group of requirements are induced by the software engineer and the working environment. The techniques have to reflect language and point of view of the software engineer. Otherwise one struggles with the technique's way of expressing terms and cannot focus on the primary work. The MidArch method is also used to make knowledge explicit (see 2.1) so that engineers can learn from the experience of others. This implies that different persons in different projects and domains contribute descriptions for the knowledge base. This input could result in information which is contradictory or at least different in terminology which lead to disjoint isles of knowledge. To avoid such incompatibility the technique should support the interconnection of these isles. Therefore four additional requirements can be derived:

Requirement 5 *The language and view of the software engineer must be the basis of the vocabulary used to codify information.*

Requirement 6 *The knowledge modeling technique must follow the style modeling procedure to avoid context switches during style development.*

Requirement 7 *The knowledge modeling technique should avoid the forming of isles of knowledge.*

Requirement 8 *The knowledge modeling technique must interconnecting of isles of knowledge.*

4.2 Basic Modeling Concept

In this section, the basic concept of the knowledge modeling technique is developed. The framework for this technique is the MidArch style modeling procedure and therefore the technique follows the five steps of the procedure. The requirements 1 and 6 (see above) are fulfilled by using the MidArch style modeling procedure as a framework.

Based on the steps of the modeling procedure, the associated additional tasks for the analysis technique are sketched. In the following, these tasks are briefly introduced.

Informal Modeling In this first step information is gathered from technical documentation, like: middleware platform documentation, system documentation, or application documentation, and experts of these areas. For the documentation a questionnaire and a paraphrasing method can be used to extract meta-information. These experts are best interviewed by predefined question catalogs. Experts can also serve as early reviewers. The resulting meta-information forms the informal knowledge description and becomes part of to the informal style description.

Relating In the second step, the new informal style is related to existing styles and technologies. For example: If a middleware-framework is Java-based, then this relationship must now be recorded. Therefore the knowledge base is interrogated for existing technology definition (see also section 3.3) and if necessary new definitions are created for later addition.

Codification Out of previously defined informal style meta-information, more structured sentences are derived. The syntax of these sentences should be close to OWL and SWRL formalisms. Used objects in the informal style description should be categorized by existing, or if necessary, additional concepts, which have to be expressed explicitly.

Formalization In this modeling step, the structured sentences from the codification step are transformed in OWL classes, individuals, and properties and in additional SWRL rules. The formal style descriptions are used to define the corresponding structures in the knowledge base. This means each style is represented by an *ArchitectureFamily* or *MiddlewareOrientedArchitectureFamily* individual and all component, connector, port, and role types are represented by individuals of the their classes respectively.

Reviewing The last step is dedicated to a review process. In the MidArch style modeling procedure the formal style is checked for consistency. Analog, the meta-information is checked for completeness. Additionally relationships of the newly formed knowledge with other, stored previously meta-information have to be evaluated. The review process is governed by requirement 8.

4.3 Detailed Modeling Concept

The previous section gave an overview of the knowledge modeling technique and explained the relation to the MidArch style modeling procedure. Section 4.3 focuses on the detailed design of each task. In figure 4.1 the technique is shown as a diagram. The names on the left represent the five steps of the MidArch style modeling procedure as a reference for the proposed knowledge modeling technique.

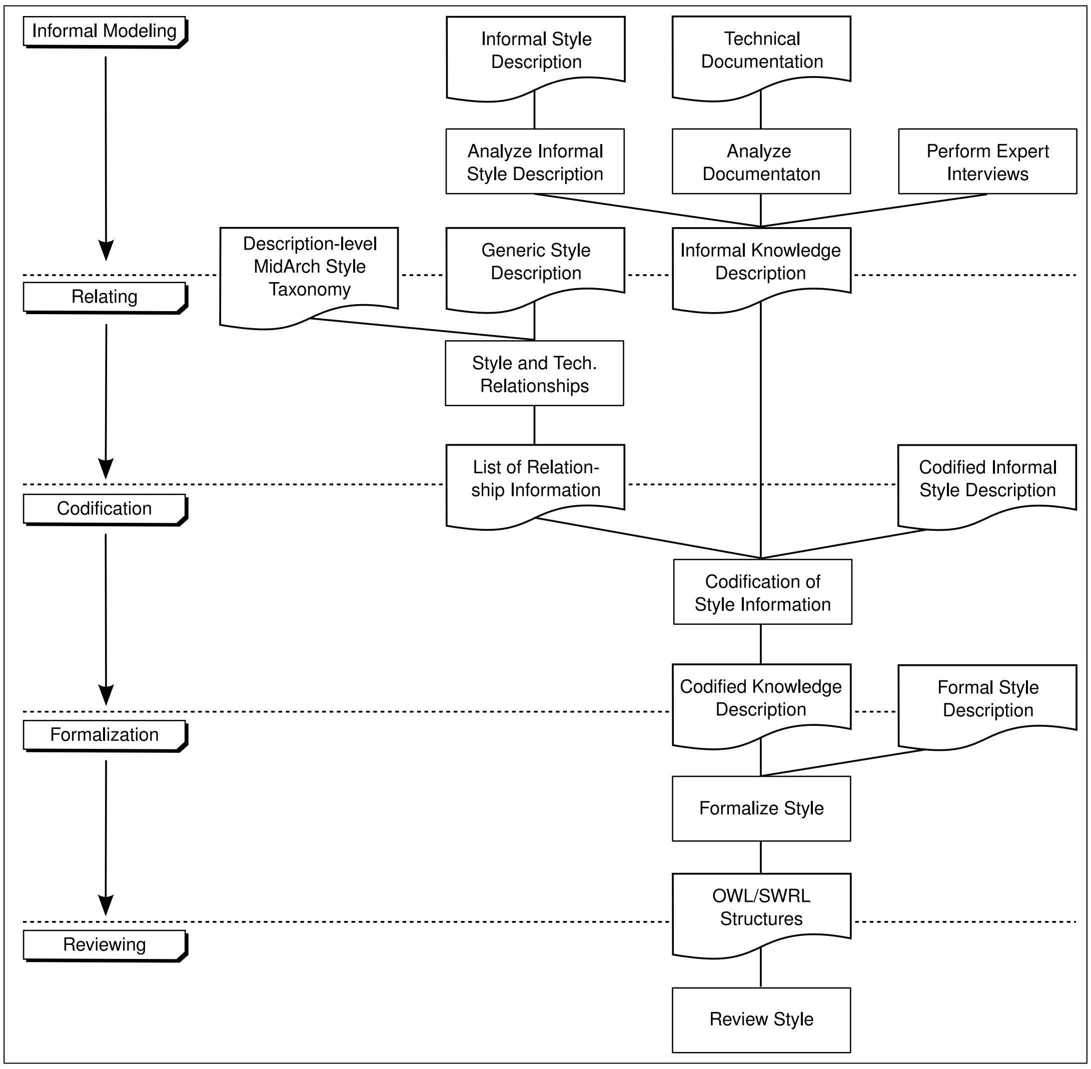

Informal Modeling
Informal Style Description
Technical Documentation
Analyze Informal Style Description
Analyze Documentaton
Perform Expert Interviews
Relating
Description-level MidArch Style Taxonomy
Generic Style Description
Informal Knowledge Description
Style and Tech. Relationships
Codification
List of Relation-ship Information
Codified Informal Style Description
Codification of Style Information
Formalization
Codified Knowledge Description
Formal Style Description
Formalize Style
Reviewing
OWL/SWRL Structures
Review Style

Figure 4.1: The knowledge modeling procedure, its tasks, and artifacts based on the steps of the MidArch style modeling procedure (see figure 2.5).

To illustrate the knowledge modeling technique a simplified example of the *Spring Framework* is used, which has already been used in section 3.2 to introduce different ontology modeling approaches. As an example input text the following sentences are used.

> Spring supports the Model-View-Controller design pattern through JSF [Sun Microsystems 2008b]. The Spring Framework implements the Inversion of Control pattern.

The subsections below describe the tasks of the knowledge modeling technique, their input and output documents, their relationship to the style modeling procedure, and the methods to collect and assemble knowledge. Subsection 4.3.1 describes the informal modeling where new information is introduced into the knowledge building process. Subsection 4.3.2 explains how new information is related to existing information. In subsection 4.3.3 the informal information is transformed into a standardized description. The formalization of the new information is explained in subsection 4.3.4. And finally, subsection 4.3.5 sketches the review for the knowledge modeling technique.

4.3.1 Informal Modeling

This first step in the knowledge modeling procedure is used to introduce new information to the knowledge process. This is done by examining three different input sources as shown in figure 4.1. In the absence of a platform expert, the interview task can be skipped. However the analysis of the platform documentation and the informal style description are required.

To explain the analysis of these input sources, the knowledge to be retrieved must be characterized. This subsection is therefore subdivided in four parts. Three knowledge aspects are defined first, which describe characterization of the knowledge. Second, the analysis of the informal style description is explained. Third, the analysis of technical documentation with a paraphrasing technique is discussed. And finally the optional and supplemental interview task is introduced.

Knowledge Aspects

The style composition, its component and connector types are provided by the style modeling procedure and are therefore already acquired and present in the informal style description. Therefore the analysis of the input documents can focus on three knowledge aspects which represent three different views on a middleware-oriented style. The first aspect relates to the *target domain* and the domain vocabulary of the middleware-oriented style. The second aspect focuses on *technical aspects* of the style. And the third aspect deals with the *quality aspects* of the technology and its documentation.

1. The software engineer or style developer examines the *target domain* of the analyzed platform technology and identifies the problems the platform developers had in mind when creating the platform. Additionally documentation of the application domain have to be analyzed and the resulting information has to be captured.

2. The *technical aspect* focuses on information of supported design pattern, used technologies, or related frameworks. For example the previously mentioned *Spring* style supports various design patterns (see section 3.2). The related technology is known to work with other frameworks to implement these patterns. It is also important to know what dependencies Spring has. For example: Spring is Java-based and therefore works good with other Java-based frameworks, however to use it with Python or C introduces additional cost and complexity.

3. The suitability of a platform is affected by *quality aspects* like interoperability and stability. Additionally the support for the platform by a vendor or a community is also an important quality aspect, because unsupported software can potentially result in higher cost.

Analyze Informal Style Description

The informal style description from the *Informal Modeling* step of the MidArch style procedure contains structural information about the resulting styles and meta-information about the style and its building blocks. The description is normally composed of a section for the style overview as well as sections for the styles component, connector, port, and role types.

Initially this structural information is extracted and used to define categories in the resulting *informal knowledge description* of the style meta-information. After that, the informal style description is evaluated with the three, above defined knowledge aspects in mind.

Firstly, the target domain of the technology and its vocabulary is determined by reading the description. Secondly, information about supported technologies and design pattern is retrieved, if any have been mentioned in the description at all. And thirdly, the description is scanned for quality aspects. However the informal style description, as present in [Bornhold 2006], is focusing only on structural information and domain knowledge. Therefore not all informal style descriptions contain information for all three knowledge aspects.

Analyze Technical Documentation

The platform or technology documentation can be one or more documentation artefacts such as: specifications, tutorials, examples, or lists of frequently asked questions (FAQ). These documents normally provide answers to all three knowledge aspects, however the information is scattered across such documents. Therefore the first step is to identify and obtain all such documentation. Then the documentation is analyzed and categorized into the three knowledge aspects.

A possible working analysis method is reading the documents in a structured way by paraphrasing paragraphs. This means, each paragraph is subsumed in one sentence. Transitional and introductory paragraphs are skipped. However, this would lead to many meaningless sentences. A preselection of relevant sections and subsections for analysis is recommended.

To speed up the process, the use of categories is useful. Categories help focusing on important information. However categories have to be created. In the content analysis literature [Gerbner et al. 1969, Mayring 2000] two category generation approaches are described. The first approach is called *inductive category development* and the second is *deductive category application.* The main difference is that the inductive approach starts with a criterion for selecting categories. This means, it describes when to form a new category and what level the category should be. The deductive approach defines categories on the basis of an underlying theory for the research [Mayring 2000]. Used to analyze technical documentation, this theory would be a description of meta-information of an architectural style.

In case of technical document analysis for meta-information, a theory-based approach is less useful, because predefined categories do not exist. Furthermore the deductive approach for finding knowledge is more applicable in order to test a theory, than to collect new knowledge for building a new theory. Theory building is better done with the inductive approach, because it is designed to gather new knowledge.

The inductive category development [Mayring 2000] is a step based approach (see figure 4.2). Step 1 is the definition of an initial research questions. For my thesis, the three knowledge aspects provide these research questions.

In step 2 (see figure 4.2), the selection criteria for categories are defined on the basis of the research questions. The development of a good criteria is a complex task. As a start, a rough criterion is defined, which can be refined through later feedback. While using this modeling technique, a better understanding of the problem develops, which in return is used as feedback for the criteria development [Mayring 2000]

As primary selection criteria, I propose the three knowledge aspects. Additionally existing concepts in the knowledge base can also be used as initial categories and serve therefore as second level selection criteria. In general, everything which is represented by a concept can be labeled as a category. This leads to three alternative criteria which can be used in the selection:

1. Select or create categories by knowledge aspect
2. Every definition in the document, which is a concept should be used as a category
3. All existing concepts from the knowledge base shall be used as categories

The third step (see figure 4.2) of the inductive approach is used to formulate categories. In this process the technical documentation is partially read. The literature [Gerbner et al. 1969, Mayring 2000, Krippendorff 1980, Rosengren 1981] does not specify a method for selecting the relevant parts for the analysis. This must be done by the software engineer.

For example, most documentations have some sort of introduction chapter or section which discusses the motivation and use of the technology. Therefore such a chapter can provide material for categories for the target domain. FAQs are also a good source for category information, because information, engineers often need to know, are frequently asked and therefore collected in FAQs.

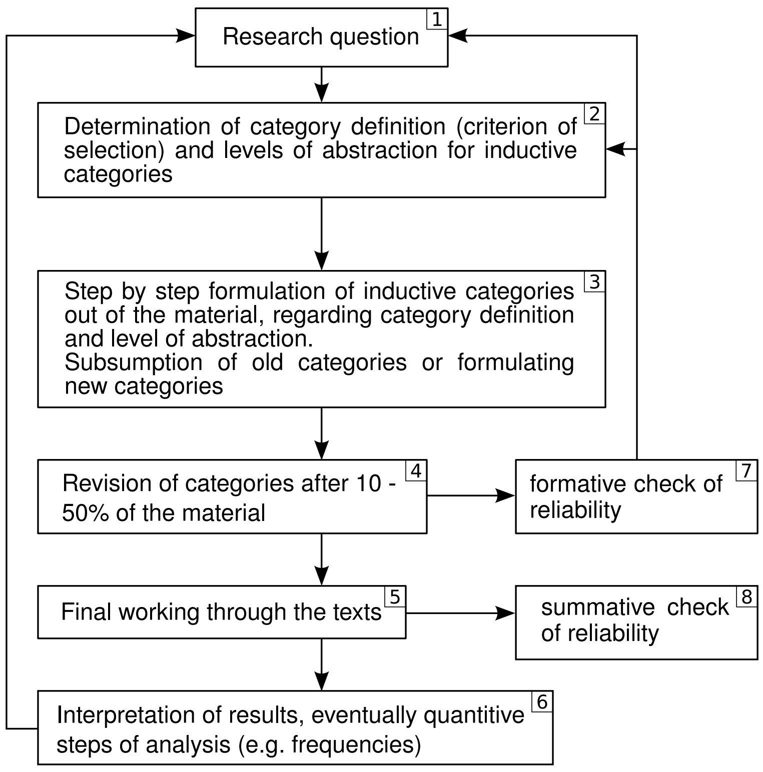

Research question 1
Determination of category definition (criterion of selection) and levels of abstraction for inductive categories 2
Step by step formulation of inductive categories out of the material, regarding category definition and level of abstraction. Subsumption of old categories or formulating new categories 3
Revision of categories after 10 - 50% of the material 4
formative check of reliability 7
Final working through the texts 5
summative check of reliability 8
Interpretation of results, eventually quantitive steps of analysis (e.g. frequencies) 6

Figure 4.2: Chart showing the inductive category development [Mayring 2000].

The method described by Mayring [2000] works in an incremental way. Therefore only 10% of the selected documentation are analyzed. Then the first revision of the categories is undertaken (step 4 and 7). If the categorization up to this point proves unsatisfactory, then the selection criteria have to be revised, and the selected texts have to be re-examined. After this first revision more text is analyzed for categories and if necessary the categories are revised again.

This category building and revising loop is repeated until a satisfactory set of categories emerges. Mayring [2000] states that not more than 50% of the text should be analyzed in this way. For long documentation even less text might be sufficient. The categorization can be reused for similar documents for the same or other technologies.

After building the categorization the text is paraphrased. In the step 5 (see figure 4.2) all selected text parts are processed by paraphrasing. Paraphrasing is a technique where sentences from the original document are reformulated in shorter sentences, clauses, or even phrases with more standardized expressions by the reader [Strobl 1996].

Alternatively, a technical way to formulate sentences is a statistical identification technique for key sentences [Coenen et al. 2007]. Such algorithms are used in text classification and preprocessing, which are used in machine learning. However, these algorithms need keywords and therefore training documents. Therefore in a bootstrapping phase textual analysis is required.

All sentences collected in the documentation analysis task are assigned to the already established structured list in the informal knowledge description. While examining different documents similar sentences might come up. These similar sentences must be merged and duplicates have to be eliminated. The resulting sentences are stored in the structured list while the obsolete sentences are removed.

The interpretation of the result step (step 6) in figure 4.2 is, in my method, not part of the content analysis. It is done in the *Relating* and *Codification* task explained in section 4.3.2 and 4.3.3

Expert Interview

The last task in the *Informal Modeling* step is interviewing platform experts. In this task the gathered knowledge from the previous tasks is presented to the platform expert. The three knowledge aspects have to be addressed in the interview with the platform expert. The expert can then verify the collected information, correct mistakes, and add missing facts. As result of the interview the final informal knowledge description is created, which is later used in the codification of the style's meta-information.

4.3.2 Relating

The *Relating* step is used in the style modeling procedure to find relationships of existing styles in the style repository. The knowledge modeling technique extends this step by finding relating properties, technologies, and design pattern. All these relationships are compiled as a list of relationships.

Considering the example: The Spring Framework implements the Inversion of Control pattern. The sentence indicates a relationship between *the Spring Framework* and the *Inversion of Control* pattern. From the knowledge base the information can be retrieved that *Inversion of Control* is defined as an *ArchitectureFamily* (see section 3.5.2). Therefore the relationship list contains the entry *Spring* is a specialization of *Inversion of Control.*

Beside the MidArch relationships of variation and specialization, the knowledge base can also express exclusions or complements. Exclusion means here, that a certain platform or framework does not work together with another framework. And complement means, a technology or framework complements another framework to make a whole. Because the knowledge base is defined as an open system, other relationships can be introduced as well and therefore they should be added to the list of relationships.

4.3.3 Codification

The two initial tasks focus on collecting information and meta-information for a style. In the *Codification* task, the information is mapped to the MidArch style meta-model. The information collected for the knowledge modeling technique (meta-information of the style) has to be mapped to the ontology constructs as well.

The sentences stored in the informal style description and the relationship information are analyzed to determine noun phrases (see section 2.1.3). From the example: Spring supports the Model-View-Controller pattern through Java Server Faces [Sun Microsystems 2008b].

The terms *Spring*, *Model-View-Controller* pattern and *Java Server Faces* are subjects or objects. The difference between the subject and object is that the subject is the acting substantive and the objects are passive (see also section 2.1.3). However for the analysis the subject is the origin in the relation while the object is the recipient.

In the example sentence, there are different relationships. First, Spring *supports* Model-View-Controller. Second, the Model-View-Controller pattern *can be implemented with* Java Server Faces. And third, Spring *works with* Java Server Faces. To identify these relationships, the sentences have to be broken down in simple subject predicate object sentences. All subjects and objects found by this analysis task can be considered individuals.

The next task is to find suitable classes for the determined individuals. Therefore existing classes have to be examined. For styles, technologies, and design patterns, the classes defined in chapter 3 have to be used. However, sentences might introduce new concepts and therefore also new classes have to be defined. Especially the target domain of the platform or the application domain of the MidArch instance are sources of new concepts.

Similar to the class finding task, the retrieved relationships have to be mapped to newly created properties, but preferably to existing ones. Therefore the relationships have to be compared with existing property definition. In some cases relationships are expressed as the inverse of an existing property. In that case the relationship must be inverted.

In the above example the relationship, *can be implemented with*, relates design patterns with technologies. However the knowledge-base already has a property relating technologies and design patterns called *supportsPattern*. Therefore the *can be implemented with* relationship can be expressed with this existing property.

Beside classes, individuals, and properties, general rules can be extracted from the informal description. The rule modeling is however a more complicated task. This is so, because the engineer cannot just look for syntactical elements in the sentences, but has to understand the sentences and check if they express a general rule. Also it might be necessary to look at more than one sentence to see such rules.

However, after identifying one or more sentences, which describe a rule, linguistic objects in the sentences are mapped to individuals or classes from the knowledge base. As explained in section 2.2.3, classes are represented by unary predicates in SWRL. The linguistic predicates are mapped on binary SWRL predicates, which are OWL object or datatype properties.

If the sentences include conjunctions, like *and*, *but*, and *or*, they must be translated into logical operators. The conjunction *and* is very simple to transform, because it can directly be mapped to $\wedge$ (logical and) in a rule. The conjunction *or* however is more complicated. The general idea is, that a new rule is specified for each alternative predicate or sequence of predicates. In the simplest situation, if there are only two logic predicates and they are connected with *or* then two rules have to be formed. One for the left predicate and one for the right predicate. The last conjunction mentioned here is *but*. It can be interpreted as a negation of all predicates in the sentence following the conjunction. Therefore the *but* is removed and all predicates are negated.

Beside the already explained linguistic elements, there are more constructs. However, a detailed analysis of natural language clauses and their transformation in horn clauses is not subject of my thesis, but can be found in [Dale 2000, Jackson and Moulinier 2002].

As result of the *Codification* task, an informal codified knowledge description list of classes, properties, individuals, and rules exists. This list must be checked for consistency. The criteria for the consistency check are that the model of architecture families introduced by the knowledge-base have to be honored.

4.3.4 Formalization

The *Formalization* task is the last creation task and comprises the formalization of the informal description of classes, properties, and individuals from the *Codification* task. The formalization can be performed in a four step approach. First, the structure of the style is formalized. Second, new classes and individuals are implemented. Third, the new properties are declared and the relationships to the style are defined. And fourth, general rules are formulated and defined.

Style Structure Formalization

In the initial step, the style structure is described by one instance of *MiddlewareOrientedArchitecture*, with sets of *CompontentType*, *ConntectorType*, *PortType*, and *RoleType* instances assigned to it in respect to the definition of styles explained in section 3.3.1. The listing 4.1 shows a template for the description of the structure.

```
Individual(STYLE type(MiddlewareOrientedArchitectureFamily)
   value(hasComponentType COMPONENT-TYPE)
   value(hasConnectorType CONNECTOR-TYPE)
)

Individual(COMPONENT-TYPE type(ComponentType)
   value(hasPortType PORT-TYPE)
)
Individual(CONNECTOR-TYPE type(ConnectorType)
   value(hasRoleType ROLE-TYPE)
)

Individual(PORT-TYPE type(PortType)
   value(acceptsRole ROLE-TYPE)
)
Individual(ROLE-TYPE type(RoleType))
```

Listing 4.1: Template of a complete initial style definition for the knowledge base. Names in uppercase are placeholders.

Classes and Individual's Formalization

The next step is to formalize new classes and their individuals. This can be done by enumerating individuals in the class definition or by defining classes and individuals separately. Which method to choose depends on the concept to express. If their is potential that class extension (see subsection 2.2.2) might change in the future or the individuals have properties, then it is easier to implement and maintain the concept and its individuals when they are declared separately. Therefore enumerations are only suitable for concepts, where it is enough to have the individuals declared rather then defined.

Besides identifying new individuals for new classes, their might be individuals to be assigned to existing classes. These individuals must be added in a suitable way. If the class is enumerated, a new individual must be added to the enumeration, and if the class is defined as a partial class, then the individual is added by explicit definition.

Property Formalization

When all individuals and classes are formalized, new object and datatype properties can be declared. It is important to define properties as exact as possible in order to be able to obtain accurate information. Therefore the features of each property have to be investigated. As explained in section 2.2.2, properties can be functional, inverse functional, transitive, and symmetric. These characteristics can have important effects on reasoning.

For example, if a property is not transitive then the added relationships just exist. But when it is transitive, the reasoner can create new relationships based on the explicit information in the property. Additionally, properties can be sub-properties of existing properties. This aspect has also to be investigated in the formalization task, as it can lead to new knowledge.

After the definition of datatype and object properties, the individuals can be completed by assigning values to properties belonging to the respective individuals. In listing 4.2 a template for an object property definition and the assignment of a property to an individual is given.

Listing 4.2: Template of an object property definition and its use in an individual.

```
ObjectProperty(PROPERTY-NAME domain(DOMAIN-CLASS) range(RANGE-CLASS))

Individual(INDIVIDUAL-NAME type(DOMAIN-CLASS)
   value(PROPERTY-NAME VALUE)
)
```

Rule Formalization

To complete the formalization task, the codified rules are transformed into SWRL rules. In the codification task the sentences have already been broken down into predicates and individuals. Conjunctions have been identified and their logic counterparts have been associated with them. All linguistic objects and predicates have been mapped to their corresponding OWL and SWRL constructs. Now in this rule formalizing step, the sentences are transformed into horn clauses and finally in SWRL and auxiliary OWL expressions.

First, each sentence is transformed into a horn clause [Horn 1951] (see also 2.2.3). This means, the predicates forming the antecedent are being located to the left side of the implication ($\rightarrow$) and a the single consequent predicate is placed on the right side. If the sentence contained alternative conjunctions, such as *or*, then the antecedent must be broken down in several different antecedents which lead to several different rules.

For example, an antecedent contains the expression $\textit{supportsTechnology}(?x,\textit{Hibernate}) \vee \textit{supportsTechnology}(?x,\textit{JPA}) \wedge \textit{supportsPattern}(?x,MVC)$. Then this must be broken down into two separate rules, such as:

$\textit{supportsTechnology}(?x, \textit{Hibernate}) \wedge \textit{supportsPattern}(?x, MVC) \rightarrow \textit{WebService}(?x)$
$\textit{supportsTechnology}(?x, \textit{JPA}) \wedge \textit{supportsPattern}(?x, MVC) \rightarrow \textit{WebService}(?x)$

As already mentioned in section 2.2.3, SWRL is not able to handle negations. So rules requiring negations should be noted in a comment to the architecture style description, so they can be processed when more powerful rule languages are available. However some negations can be implemented with a workaround. If the negation is on individuals, stating that a rule applies when a individual does not belong to specific class, then this can be done with the help of OWL. Therefore the engineer defines a complement class for a class where the individual must not belong to. The listing 4.3 shows an example template for modeling negations.

```
Class(Interpreter partial)

Implies(
   Antecedent(!Interpreter(?x))
   Consequent(hasPrototypingQuality(?x, low))
)

Class(NotInterpreter complete complementOf(Interpreter))

Implies(
   Antecedent(NotInterpreter(?x))
   Consequent(hasPrototypingQuality(?x, low))
)
```

Listing 4.3: A possible workaround for SWRL's limitations in handling negation.

The first rule expresses that a language which cannot be interpreted but only compiled is not adequate for prototyping. However the rule includes a negations, so the second definition works around this problem. The class *NotInterpreter* is defined as identical (complete) to the complement of *Interpreter*.

4.3.5 Reviewing

The final step of the MidArch style modeling procedure is the review process. Analog to this modeling procedure, the knowledge modeling technique has also a review task at the end. While some reviewing of the initial information is already done in the first task and some consistency checks are performed in the formalization task, this final task allows to check if the new concepts, individuals, properties, and rules are really appropriate for the described style.

Other styles might benfit from newly introduced concepts and properties. Therefore the styles have to be reviewed with these new concepts and properties in mind. This review process minimizes possible fragmentation tendencies in the knowledge base.

5 Knowledge Querying Technique

The *knowledge querying technique* is the counterpart to the modeling technique from the previous chapter. While the modeling technique is used to extract information from various sources and store them in a knowledge base, the querying technique is used to get information from the knowledge base.

This chapter discusses the knowledge querying technique. First, the context for the technique and the requirements are explained. Second, a process for the technique is sketched. And finally, the tasks for the process are specified.

5.1 Context and Requirements

The context of the *knowledge querying technique* is the style selection process of the MidArch method (task 3B) [Giesecke 2008, sec 9.4]. This process works hierarchically on the MidArch style taxonomy (see section 2.4.2) with evaluation data from previous MidArch instances. The querying technique can augment this process in two possible ways.

First, it allows preselection of styles without analyzing their evaluation data. This selection is helpful with styles, which have insufficient evaluation data. Second, requirements which are represented by nominal or binary values, as well as requirements which cannot be represented by any quality model, can be expressed as queries for a knowledge base as long as they refer to known concepts and individuals.

In the MidArch method, the style selection process, and the knowledge base induce a set of requirements for the querying technique. The requirement documentation of a given MidArch instance, as specified in task 1B of the MidArch method, is one input for the query forumulation. The other input is the quality model, because the quality model includes a set of questions, which can at least partially be answered with the knowledge base. Therefore the following requirement for the querying technique can be formulated:

Requirement 1 *The technique must be able to transform requirements and quality model questions into queries.*

Because the querying technique is used for the preselection of MidArch styles, it must support the language of the project. This means, the engineer should be able to use the vocabulary of the project for the queries or at least it should be easy to convert the language of the project into the language understood by the knowledge base. Furthermore the query formulation process should be simple in order to handle so that many queries can be formulated in short time. This leads to two additional requirements:

Requirement 2 *The querying technique must utilize the language of the project.*

Requirement 3 *The query formulation process must be simple.*

5.2 Basic Process Design

The *knowledge querying technique* is used in the MidArch task 3B [Giesecke 2008, sec. 9.4] as an additional method to select style candidates. While the primary method in task 3B implies a hierarchical search approach through the taxonomy of existing styles, the knowledge querying technique works differently. Based on requirements and questions from the quality model, the technique queries the knowledge base for suitable style candidates.

The querying technique can be divided in five steps. First, relevant information is extracted from the list of requirements and optionally from the questions already formulated for the quality model. Second, the information is analyzed for present linguistic objects and predicates. Third, based on the objects and predicates found in the previous step, horn clauses are formulated and augmented with SQWRL predicates to get useful results. Fourth, query clauses are send to the system. And fifth, the result is checked using the question and query sentences from the first step. If the results are good enough to select a style, then the process ends. Otherwise the queries have to be refined and the codification, formalization, and querying steps have to be repeated.

Question Retrieval The term *question* is, in this context, not limited to its linguistic definition. Additionally sentences or sentences which represent queries or requirements, like *we need a Java-based style*, are also considered questions, because they ask for some specific property of the system.

Codification The codification step is used to determine objects and predicates from the previously selected sentences and to find corresponding individuals and properties in the knowledge base. These individuals, and properties are the basis of the codification result. To make the codification process complete the relation between individuals and properties have to be saved.

Formalization With the codified sentences from the previous step a set of horn clauses can be defined. Because horn clauses are disjunctions of literals, they can be combined to make a query more specific. Additionally SQWRL predicates can be added to further retrieve, sort, or manipulate the result (see section 2.2.3).

Querying In the querying step the query clauses are send to the query interface of the knowledge base and the results are logged for later analysis.

Selection and Refinement The results from the querying step are analyzed using the question sentences from the first step. If the results are sufficient one or more styles are selected as style candidates. If not, the queries are refined and the steps starting with codification are repeated.

5.3 Detailed Process Design

The previous section gave an overview of the knowledge base querying technique. This section focuses on the detailed design of each step. Figure 5.1 shows the steps and the involved artifacts. The dotted lines represent data flow and the solid lines indicate process flow.

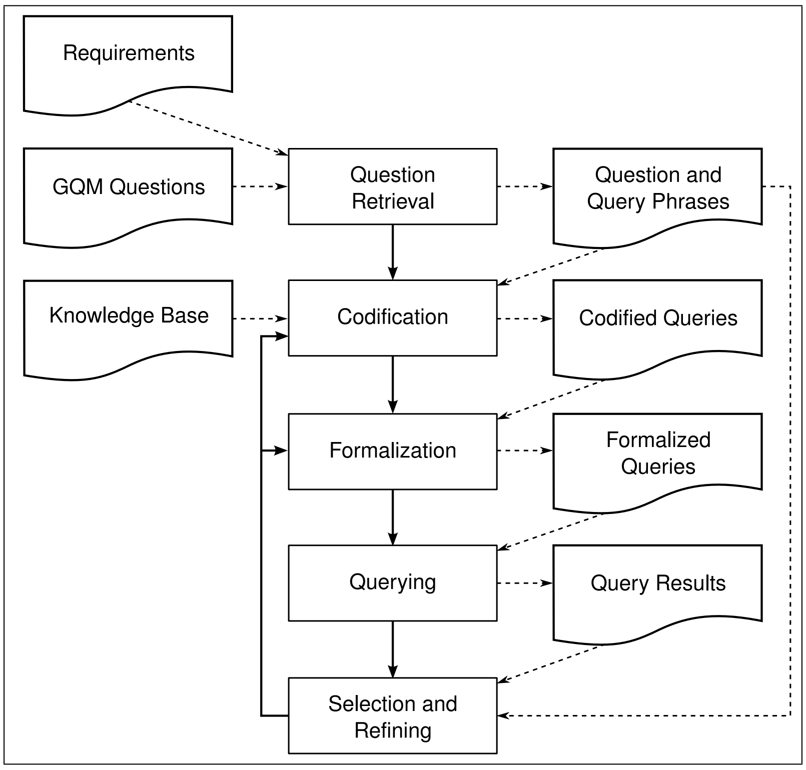


Figure 5.1: The knowledge query technique, its tasks, and artifacts.

The following sections describe the steps of the knowledge querying technique shown in figure 5.1, and the corresponding input and output documents.

5.3.1 Question Retrieval

The *question retrieval* step has similarities to the informal modeling step of the modeling technique. Its purpose is to collect information. However in the context of the query technique, the information queried for are requirements from MidArch task 1B, and questions, especially questions from the GQM method [Basili et al. 1994]. Therefore the requirement document and the GQM questions are the two input sources for this step.

The engineer collects these requirements and questions and reformulates them into question and query sentences. These sentences are then processed by a paraphrasing technique [Strobl 1996] to remove redundant sentences. This paraphrasing is important because requirements and questions may address the same issue and would therefore only bloat the later codification.

The result of this step is a document containing only only valid and unique query and question sentences.

5.3.2 Codification

The query and questions sentences from the retrieval step are processed in the *codification* step to obtain structures which are close to horn clause like query rules. Therefore the engineer transforms the natural language sentences to predicates and individuals. To do so the engineer analyzes the sentences for linguistic objects and predicates.

The linguistic objects are then mapped to individuals or classes from the knowledge base. As explained in section 2.2.3, classes are represented by unary predicates in SWRL. The linguistic predicates are mapped to binary SWRL predicates, which are OWL object or datatype properties.

Similar to the knowledge modeling technique (see section 4.3.3), conjuntions like *and*, *but*, and *or* must be translated into logical operators. The conjunction *and* can directly be mapped on $\wedge$ in a rule. The conjunction *or* is more complicated to process, because it has to be resolved. However the resolving is done in the formalization task. Therefore the conjunctions are just marked. The last mentioned conjunction is *but*. It can be interpreted as a negation of all predicates in the sentence following the conjunction. Therefore the *but* is removed and all predicates are negated.

5.3.3 Formalization

In the previous step, the sentences have been broken down into predicates and individuals. Based on conjunctions, sentences have further been split into several separate codified sentences. All linguistic objects and predicates have been mapped to their corresponding OWL and SWRL constructs. In the *formalization* step these sentences are now further processed.

First, each sentence is transformed into a horn clause with an empty consequent. Clauses based on alternative sentences (*or* conjunction) are still marked as alternatives, because this information is relevant for the last formalization step. Second, negations have to be resolved. And third, the clauses have to be combined.

As mentioned in section 2.2.3, SWRL is not able to handle negation. Therefore the workaround mentioned in section 4.3.4 must also be applied. Therefore unary predicates, which symbolize classes, can be negated by the definition of a complementary class in OWL. For binary predicates however, there is no workaround. As long as RIF-BLD [Boley and Kifer 2007] is not available there is no solution. Consequently negated binary predicates have to be dropped in the formalization. However their meaning should be recorded for the *selection and refinement* step.

The last step in *formalization* is the combination of clauses. All clauses, which have to be satisfied together, are concatenated into one larger clause. For example, if the resulting style should be Java-based and Spring-based, then these two clauses are concatenated.

For alternative clauses, there must be one resulting clause per every alternative clause. For example, if there are two sets of alternative clauses $A = \{A_1, A_2, A_3\}$ $B = \{B_1, B_2\}$ and one additional clause C then this results in six clauses.

$$\begin{aligned}
&A_1 \wedge B_1 \wedge C \rightarrow \\
&A_2 \wedge B_1 \wedge C \rightarrow \\
&A_3 \wedge B_1 \wedge C \rightarrow \\
&A_1 \wedge B_2 \wedge C \rightarrow \\
&A_2 \wedge B_2 \wedge C \rightarrow \\
&A_3 \wedge B_2 \wedge C \rightarrow
\end{aligned}$$

These resulting clauses are written in SQWRL rules. Depending on the initial intention of the query, different SQWRL predicates can be used. In most cases *sqwrl:select* is the right choice, because the engineer wants all matching individuals. In some rare cases an ordering with *sqwrl:orderby* might be helpful or the selection of the maximum, especially when measurement values are part of the query.

5.3.4 Perform Querying

In the querying step, previously defined queries based on formalized SWRL rules are send to a reasoning and rule engine. The results are collected and stored for later analysis.

5.3.5 Selection and Refinement

This last step is used to interpret the query results. Therefore the results from the previous step are inspected and assessed. If the results are satisfactory a style candidate is selected. In the case that more then one candidate is found, all candidates can be selected as result for the method. However, if the results are inconclusive or not satisfactory, the queries are refined and the technique is repeated starting with the *codification* step. To check if the results are satisfactory, the results are compared against the relating questions and requirements from the first step of the querying technique.

6 Evaluation

Chapter 4 and 5 introduced and defined techniques for ontology-based knowledge modeling and querying of middleware-oriented architectural styles. To show that these techniques can be a useful asset in style exploration and selection for the MidArch method [Giesecke 2008], they have to be evaluated. The evaluation is performed on basis of a case study utilizing the MidArch method [Bornhold 2006].

First, the knowledge base is populated with meta-information on the styles described in the case study. Therefore the modeling technique is used on all styles described in [Bornhold 2006]. Second, the knowledge base is interrogated with queries derived from the architecture selection process and the requirements of the case study.

6.1 Evaluation of the Knowledge Modeling Technique

The case study [Bornhold 2006] was conducted to evaluate the activities and processes of the MidArch method [Giesecke 2008] on basis of an exemplary integration scenario. The case study focuses on the style development (see section 2.4.1) because no previous style descriptions from other case studies of the MidArch method were available.

The integration scenario of the case study is the regional trade information system *RegIS Online* [GmbH 2008] which is used to provide information on the economic potential to support sustainable regional development in north-west of Germany. RegIS Online consists of three independently developed subsystems. The subject of the case study was to create and evaluate different architectures for the integration of these three subsystems.

In my thesis the style modeling procedure is augmented with the knowledge modeling technique from chapter 4 which results in additional information, so called meta-information, of the modeled styles. This information is used in the knowledge querying technique for style selection process of the MidArch task 3B (see section 2.4).

To test the knowledge modeling technique the styles from the case study and their corresponding documentation are analyzed and processed with this technique (see appendix A). And the resulting meta-information is stored in the knowledge base. The evaluation results of this test are documented in the following subsections.

6.1.1 Inversion of Control Style

The *Inversion of Control* style is not a middleware-oriented style. However, it is described in the case study a an abstract basis style, because it is the basis for the *Spring* and *Avalon* styles. Furthermore, [Fowler 2004] states that many frameworks implement the *Inversion of Control* **pattern**, which is the basis for the *Inversion of Control* **style**.

As input documents for the knowledge modeling technique, the informal description of the *Inversion of Control* style from [Bornhold 2006] and the discussion of the *Inversion of Control* pattern from [Fowler 2004] were used.

The analysis for the *Inversion of Control* style produced only a few phrases and therefore only a minimal set of formal descriptions (see section A.1.4) were produced. However, these descriptions are of great importance, because they introduced basic knowledge on *Inversion of Control* styles, especially different properties of these styles.

6.1.2 Spring Style

The analyzed second style is the *Spring* style induced by the *Spring Framework*. This framework is a Java-based framework for enterprise application. It aims to be easier to use as J2EE or other Inversion of Control frameworks.

As input source for the knowledge modeling, the informal description of the *Spring Style* in [Bornhold 2006], the Spring website [Spring 2008] and the reference documentation of the Spring framework [Johnson et al. 2008] were used.

The documentation for *Spring* is comprehensive and structured in a way which allows the reader to find information quickly. Each part and chapter has an overview section which describes the technology and its purpose briefly and precise. Therefore, it was easy to obtain knowledge from this documentation with respect to the three knowledge aspects (see section 4.3.1).

Through the analysis several new concepts were introduced. These are the concepts *Package*, *License*, *Feature*, *ApplicationType*, and *Standard*. Additionally, the concept *Technology* has become the root of a technology taxonomy.

The *Spring Framework* is subdivided in packages which bundle functionalities and introduce component types. Functionality can be added to a system by including a package. Hence, the use of packages can lead to additional middleware-oriented styles. Therefore, packages and their abilities are important information for style development and description.

The *License* concept was added because it is a quality aspect for technologies and the middleware-oriented styles using technology. Some licenses restrict the use of software in certain areas or restrict the combination of technologies which would otherwise work perfectly together. Therefore it is important to know if license issues exist for a selected combination of technologies.

Many properties of styles or technologies can be described by a boolean value. For example: Does a technology support *Separation of Concern*? The answer is either ”yes” or ”no”. This boolean property approach would lead to a vast set of datatype properties. To avoid this the property *providesFeature* was introduced and a corresponding concept, which comprises all features. This property also allows to resolve connections between technologies and styles.

The last two concepts are *ApplicationType* and *Standard*. *ApplicationType* is used to describe the target and the application domain of a style. The concept *Standard* comprises terms describing various standards. Both concepts are used to introduce vocabulary. However, *ApplicationType* is currently more a stub than structured knowledge, because the input documentation does not elaborate the topic of different application types.

The properties derived from [Shaw and Clements 1997] did not play a great role in modeling the knowledge of the *Spring* style, because *Spring* does not imply restrictions in the area of topology, data intensity and flow. The properties introduced by the *Inversion of Control* style, however, were greatly used to model knowledge on the *Spring* style.

6.1.3 Avalon Style

The *Apache Avalon Project* [Avalon 2004] is the source of the Inversion of Control framework *Avalon*. Since 2004 the project is closed and the *Avalon Framework* is now part of the *Apache Excalibur Project*.

The input source for the knowledge modeling are: the informal style description from [Bornhold 2006], the sections *Fortess* and *Framework* which where sourced from the web-page of *Apache Excalibur* [Excalibur 2007].

The documentation of the *Avalon Framework* is far less detailed and complete compared to the *Spring Framework* documentation, which made it much more difficult to gather information on the *Avalon* style in respect to the three knowledge aspects.

The *Avalon* style was the second *Inversion of Control* based style to be analyzed. Therefore properties and individuals from the previous analysis of the *Spring Framework* have been reused in the knowledge modeling for*Avalon*. Especially in the relating task several relationships to previously defined individuals where found which were not introduced by the documentation of *Avalon* directly.

The codification task, revealed that *Avalon* uses a different vocabulary to introduce its own concepts than *Spring*. Therefore a mapping of this vocabulary to the vocabulary used by *Spring* and the underlying knowledge base was added. For example: The concept *PortType* from the knowledge base expresses the same concept as *Interface* in the *Avalon* documentation. Therefore the term *Interface* was introduced and declared to be a synonym. Such synonyms help the engineer to find knowledge, because the engineer search for interface or port type.

The final reviewing task consisted of checking the new definitions for consistency, and the existing individuals were compared to new concepts and individuals introduced by the analysis of *Avalon*. This reviewing task resulted in an update of previously defined individuals such as *Hibernate* (see section A.3.5).

Additionally to new relationships, between newly formed individuals and existing ones, new rules were introduced to support the inheritance of properties throughout a style. For example: If a package provides a feature, than it is obvious that the whole style provides the same feature. However, the inheritance of features was not defined previously, but the reviewing task revealed this problem.

6.1.4 Cocoon Styles

The *Cocoon* styles are *Pipes-and-Filter*-based styles. The corresponding abstract style is introduced in section 3.5. The two styles described in [Bornhold 2006] are very similar. The difference between the styles is that the first style, called *Cocoon Basic Style*, does not use *FlowScript*, where the second style, called *Cocoon with FlowScript*, does. Therefore the second style is an extension of the basic style.

Because of the close relationship of these two styles, the analysis and formalization were done together. The input documents were the Cocoon web-site [Cocoon 2008] and the style description from [Bornhold 2006].

Unlike the *Spring* and *Avalon* frameworks, *Cocoon* is not an *Inversion of Control* style. Technically *Cocoon* uses the previously mentioned frameworks and the *Inversion of Control* properties to realize a *Pipes-and-Filter* style.

The analysis of *Cocoon* resulted in many new terms, because *Cocoon* uses a different basic style and has a different purpose. Due to its underlying style *Pipes-and-Filter*, the *Cocoon* styles use properties introduced by [Shaw and Clements 1997].

The reviewing of the resulting individuals and concepts did not reveal new relationships for the *Inversion of Control* styles. However existing technologies got new property values. For example the feature *ObjectRelationalMapping* was added to the individual representing *Hibernate*.

6.1.5 Conclusion

The knowledge modeling technique worked as expected and produced many different concepts, properties, and individuals. However, certain difficulties were encountered. First, the documentation of analyzed styles and the underlying technologies vary in quality and completeness. The lack of documentation results directly in an incomplete knowledge model for the analyzed technology and style. Also the interviews with the platform experts were not performed, which had the effect that the collected sentences were not checked by an external expert. Therefore erroneous definitions might exist in the knowledge base.

Second, the different frameworks use different vocabulary to describe similar concepts. For example, *Avalon* uses the term *Interface* to describe *PortType*. In this case a direct mapping was possible. However, *Avalon* also uses the term *Concern* which describes a subset of *Interface*, but also an aspect of the pattern *Separation of Concern*. When using the technique is a strict way, this would result in a concept *Concern* which is a sub-concept of *Interface*. But this would not reflect the true meaning of *Concern*. Therefore it had to be skipped. Such decisions to evaluate terms have to be made by the software engineer using the modeling technique.

The third issue revealed during the evaluation, was that there is no clear definition of the meaning of *technology* and *standard*. For example: The programming language *Java* is a standardized language, therefore it is a standard. However, it is also classified as a technology according to [Sun Microsystems 2008c]. Another example is the concept of a *servlet engine*. The concept itself is a technology and the various implementation of this technology are instances of that concept. However, these instances are called technologies as well, according to their respective documentation [RedHat 2008, Sun Microsystems 2008a]. This results in a logical inadequacy. On one hand, *servlet container* must be addressable as an individual, and on the other hand it must serve as a concept for several implementations of this technology. As already explained, in OWL DL, a concept cannot be used as individual at the same time and vice versa. This issue could be resolved with a property allowing a technology to implement another. However, this still puts generic technology descriptions and implementations of these technologies on the same level, which is from a structural point of view unsatisfactory.

The last issue, is the missing separation of middleware-oriented style descriptions and the description of technologies. While it seamed appropriate to define the concept *MiddlewareOrientedArchitectureFamily* during the development of the initial knowledge base, in chapter 3, as an intersection of *Technology* and *ArchitectureFamily*, the processing of styles and technologies showed that such declaration results in style definition problems of the *Cocoon* styles.

While the *Cocoon* technology works with the *Spring Framework* in version 2.2 and with the *Avalon Framework* in version 2.1, this aspect cannot be described in the style itself. The two versions of *Cocoon* should result in two different styles. However, the mixing of technology and style resulted in vague knowledge for this aspect. Furthermore, the *Spring Framework* is divided into several packages, which introduce different functionality. These functionalities could result in different middleware-oriented styles. The two *Cocoon* styles are a good example for that, because one is using *Flowscript* and the other is not.

Therefore a separation of the concepts *MiddlewareOrientedArchitectureFamily* and *Technology* in the further knowledge base development would be beneficial. A style in such a refactored knowledge base would reference certain aspects, components or features of a technology and categorize these features by component, connector, port or role types.

The consequence of this evaluation is that the knowledge modeling technique, as such, works and returns processable phrases. However, the basis knowledge base needs improvements and the decision between concept and individual needs additional guidelines. Also concepts could be poorly differentiated form another in a first definition cycle. Therefore a later refinement and refactoring step for these concepts should be introduced.

6.2 Evaluation of the Knowledge Querying Technique

The previous section discussed the execution and the results of the modeling technique. In this section the knowledge querying technique is evaluated. The evaluation is based on the scenario of the RegIS Online system integration. In [Bornhold 2006] the architectural style candidates could not be retrieved from a large taxonomy of predefined middleware-oriented styles (compare MidARch task 3B [Giesecke 2008, ch. 9]), because the case study was the first use of the MidArch method. However the case study introduced four middleware-oriented styles and used two of them for candidate architectures.

In the evaluation of my knowledge querying technique requirements for the candidate architectures are used to develop queries. These queries are then executed and the results are compared to the results in [Bornhold 2006].

The development of the queries is documented in appendix B and the results are discussed in subsection 6.2.1. Finally, in subsection 6.2.2 the results of the knowledge querying technique are compared to the results from [Bornhold 2006] and some conclusions are drawn.

6.2.1 Querying

The querying process described in chapter 5 is based on five distinct steps: *question retrieval*, *codification*, *formalization*, *querying*, and *selection and refinement*.

In the *question retrieval* step, based on the target requirements (MidArch task 1B) and supplementary the goals from the quality model are collected and transformed into statements. The *codification* step is used to transform these statements in codified sentences, which are then transformed in SQWRL expressions in the *formalization* step. These expressions are executed in the *querying* step and the results are reviewed in the *selection and refinement* step.

However, the source material for this evaluation is the case study in [Bornhold 2006]. The case study does not contain a list of target requirements. Therefore, a detailed analysis of different parts of the case study was necessary to extract properties of the subject system. The process is documented in appendix B.

The compiled requirements contained, beside already defined terms, new terms for present architectural styles and technologies. Therefore these new terms had to be defined and associated with the corresponding styles and technologies. This knowledge base refinement step is not part of the knowledge querying technique as defined in chapter 5. However, it was necessary to construct more precise queries.

The rest of the of the *codification* step was then executed as defined in chapter 5 and resulted in the following list of antecedents:

1. *MiddlewareOrientedArchitectureFamily* supportsTechnology *CocoonForms* and *XSP* or *MiddlewareOrientedArchitectureFamily* supportsTechnology *Technology* providesFeature *SeparationOfViewAndLogic* and *Technology* providesFeature *FormHandling*

2. *MiddlewareOrientedArchitectureFamily* supportsTechnology *MySQL*

3. *MiddlewareOrientedArchitectureFamily* supportsTechnology *Hibernate*

4. *MiddlewareOrientedArchitectureFamily* providesFeature *FlowControl*

5. *MiddlewareOrientedArchitectureFamily* supportsProgrammingLanguage *Java*

6. *MiddlewareOrientedArchitectureFamily* providesFeature *Modularization*

In the *formalization* step, these antecedents were compiled into two final queries, because the first antecedent in the above enumeration contains an *or* conjunction which resulted in two separate alternative antecedent fragments. All other antecedents were concatenated into one antecedent, because they must be all satisfied. The compilation process in detail is documented in section B.3.

```
/* alternative rule fragments */
Implies(
    Antecedent(
        MiddlewareOrientedArchitectureFamily(?family)
        supportsTechnology(?family,XSP)
        supportsTechnology(?family, CocoonForms)
        supportsTechnology(?family,MySQL)
        supportsTechnology(?family, Hibernate)
        providesFeature(?family, FlowControl)
        supportsProgrammingLanguage(?family, Java)
        providesFeature(?family, Modularization)
    )
    Consequent(
        sqwrl:select(?family)
    )
)

Implies(
    Antecedent(
        MiddlewareOrientedArchitectureFamily(?family)
        supportsTechnology(?family,?techA)
        supportsTechnology(?family,?techB)
        providesFeature(?techA, SeparationOfViewAndLogic)
        providesFeature(?techB, FormHandling)
        supportsTechnology(?family,MySQL)
        supportsTechnology(?family, Hibernate)
        providesFeature(?family, FlowControl)
        supportsProgrammingLanguage(?family, Java)
        providesFeature(?family, Modularization)
    )
    Consequent(
        sqwrl:select(?family)
    )
)
```

Listing 6.1: The two resulting queries for *querying* step

The *querying* step produced initially unsatisfactory results due to a few missing references in the knowledge base. However, after supplying these references, the *querying* process worked as expected.

The first query (marked as alternativeA query rule) is closer to the old system described in the case study, because the technologies *XSP* and *CocoonForms* are directly linked to the *Cocoon* framework. Therefore the first alternative returned only results which contain a *Cocoon* style. The only returned solution was *CocoonWithFlowscript*, because *FlowControl* as a feature was also requested and *CocoonBasicStyle* does not have this feature.

The second query (marked as alternativeB query rule) represents a wider view. It returned *CocoonWithFlowscript* and *Spring*. The return of *CocoonWithFlowscript* was an obvious result, because *XSP* is a technology providing the feature *SeparationOfViewAnd-Logic* and *CocoonForms* is a *FormHandling* technology. In addition *Spring* was also a valid result, because *Spring* supports various *FormHandling* technologies, including one in *SpringMVCPackage*. The *SeparationOfViewAndLogic* feature is supported through *JSP* which is used in *SpringMVCPackage*.

The two results *Spring* and *CocoonWithFlowscript* represent the chosen candidate architectures for the MidArch task 3B.

6.2.2 Conclusion

In [Bornhold 2006] two middleware-oriented styles were used for candidate architectures in the case study. My method returned the same two candidates. Because of the few different style definitions in the knowledge base, this result is not surprising. Especially considering that Bornhold [2006] defined these styles directly for the purpose of style candidates for the development of candidate architectures.

However the interesting aspect of this result is, that the knowledge base returned these two styles only on the basis of a set of feature and technology requirements. Therefore the style taxonomy has not to be browsed manually which reduces the time required to find styles in the taxonomy.

7 Conclusions

The goal of my thesis is to model meta-information of middleware-oriented styles in a way that allows software engineers query this information with only the system requirements (MidArch task 1B). As a result they get a selection of middleware-oriented style candidates (MidArch task 3B) which can be used to model candidate architectures (MidArch task 3C).

Below I describe the approach used, discuss the results of the evaluation and give an persepctive on future prospects. In 7.1 the meta-information modeling approach is summarized. Section 7.2 discusses the outcome of the evaluation of the introduced modeling approach, and section 7.3 sketches future development and use of the developed knowledge base and the modeling approach.

7.1 Summary

The modeling of meta-information on architectural styles faces several difficulties. First, because the different kinds of meta-information properties of a style cannot be determined in advance, therefore these different properties have to be determined while styles are introduced to the MidArch style repository. Second, each middleware-oriented style introduces its own vocabulary based on the underlying technologies. In addition, different application domains introduce there own vocabulary.

Therefore an open and extendable meta-information modeling approach had to be introduced. The approach discussed in my thesis uses ontologies to cope with these two difficulties. Ontologies on the one hand allow the definition of terms and on the other hand ontologies allow to relate terms to another. Terms are represented by individuals (objects) and relationships are presented by predicates. In addition ontologies use concepts to classify and group individuals. This is an additional way to define properties.

As language for the formalization of the meta-information, I choose the description logic version of the *Web Ontology Language* (OWL DL). The use of OWL DL (see section 2.2.2) allows reasoning with the knowledge represented by the formalized meta-information, because reasoners, like *Pellet* (see section 2.6.2), can process OWL ontologies which conform to description logic constraints. Reasoners for OWL Full, however, do not exist. If the would, due to the undecidability of OWL Full, it could never be guaranteed that such reasoners produce any results.

One advantage of OWL DL over other description logic languages is, that it was designed with the *World-Wide-Web* (WWW) in mind. To work in this environment, the language is designed to handle contradictory definitions of terms and integrate them. It also allows to work with distributed documents, which is helpful when different engineers are using the knowledge base at different places at the same time.

To create the knowledge base, the basic concepts of middleware-oriented styles and the surrounding concepts, like *technologies* and *design pattern* were determined (see chapter 3) and transformed into classes, properties, and individuals.

With this initial knowledge base, the first styles could be analyzed and the gathered meta-information stored in the knowledge base. To do this in a structured approach, a knowledge modeling technique (see chapter 4) was developed. This modeling technique introduces a stepwise approach, first, to retrieve information from informal style descriptions and technical documentation, and second transform them into concepts, individuals, properties, and additional general rules.

While storing information alone does not serve any purpose, a query method was developed. This knowledge querying technique (see chapter 5) works similar to the modeling technique. However, it uses systems requirements from the MidArch task 1B (see section 2.4) as input and it produces query rules.

7.2 Discussion

The exemplary use of the proposed knowledge modeling and querying techniques documented in appendix A and B were evaluated in chapter 6. The evaluation revealed that the knowledge modeling technique produced the desired concepts, individuals, and properties in a straightforward way. This means, the process, layed out by the technique, worked as expected and a software engineer, who follows the process would be able to produce results in the same quality.

However, the evaluation of the modeling technique showed, that the quality of the resulting meta-information and therefore the benefit for the knowledge base is closely related to the quality of the technical documentation of the involved technologies. This quality comprises the aspects: completeness, accuracy, and accessibility.

For example, the *Avalon* documentation is very fragmented, therefore it was hard to determine were to find the information according to the three knowledge aspects (see section 4.3.1) of the modeling technique. Also the information was contradictory, because old and new documentations were mixed and it was therefore hard to determine which parts of the information is still valid.

The accessibility aspect refers here to the accessibility of information regarding the three knowledge aspects. The *Spring* documentation is therefore a good example of an accessible document in respect to the modeling technique. Every part, package, or concept explained in the documentation has its own overview section, which discusses the purpose of the part, package, or concept without going into too much detail of the implementation or configuration of these elements.

Another problem revealed by the evaluation was the definition of the concept *MiddlewareOrientedArchitectureFamily*. The concept is defined as an intersection of the concepts *Technology* and *ArchitectureFamily*. While defining the initial knowledge base this seemed to be a good approach, because an middleware-oriented architectural style is related to the concept of architectural styles as well as to the concept of technologies.

However, it was an erroneous assumption that these two relationships could be best expressed by defining the concept *MiddlewareOrientedArchitectureFamily* as an intersection of the other two concepts. The problem with this definition showed up when dealing with the two *Cocoon* styles. While there is only one *Cocoon* technology, which has different versions with different properties, there are two *Cocoon* styles referring to this one *Cocoon* technology.

Therefore a better definition of *MiddlewareOrientedArchitectureFamily* has to be based only on *ArchitectureFamily* with the extension that, middleware-oriented architecture styles, can reference a technology or parts of a technology. This allows to see a style as a distinct configuration of a technology.

The knowledge querying technique is closely related to the modeling technique not only as complementary technique itself, but also in its structure. However, the input document is a list of requirements and the result is a set of query rules.

The evaluation of this technique showed that the technique provides a well functioning stepwise approach for the query development, but it showed also that in future this technique should be improved in two areas.

First, not all projects, utilizing the MidArch method, produce separate and complete documentation of the target systems requirements (MidArch task 1B). Especially technical requirements, such as the use of certain technologies or products, are not specified explicitly in the requirements section. Therefore these requirements have to be retrieved by reviewing more documentation of the project using the MidArch method.

Second, during the query build process properties of technologies were requested, which have not been defined in the knowledge modeling task. Hence, the query development had to be interrupted and the additional information had to be added after examining the documentation of the involved technologies and styles. Only then the query development was continued. Therefore the query development's codification task should be extended by a knowledge modeling step. This step has to include: an information retrieval, a codification, and a formalization step, which could be designed similar to the knowledge modeling technique. However, the complex paraphrasing and content analysis task of the modeling technique is not required for such an extension by a knowledge modeling step of the query technique.

Beside the already mentioned improvements of the proposed techniques, the use of upcoming technologies and new techniques can be beneficial. In chapter 4 an automatic ontology learning method [Lehmann 2007] was mentioned. Such new techniques are in an early state of development, because past knowledge modeling approaches focused on rule based knowledge models. With the recent rise of ontologies due to OWL, this has become a new area of research.

Automatic learning would allow to shorten the required time for the *codification* and *formalization* tasks of the knowledge modeling technique. The paraphrasing step could also be performed by such automatic learning methods, which would allow analysing technology documentation more rapidly and therefore produce more valuable input for knowledge base in a shorter time.

Another technical improvement for knowledge modeling and querying is related to the rule language RIF-BLD [Boley and Kifer 2007]. RIF-BLD is developed for becoming the standard of *Semantic Web* rule languages in the near future. It has several advantages over the rule language SWRL and its supplement SQWRL.

The most important advantage for the modeling and querying technique is, that RIF-BLD is able to handle negation in the parameters of predicates, which allow the droping the workaround for negations, which were introduced in the modeling and querying technique. Another benefit can come from the use of formulas in RIF-BLD rules, which allows calculations and even some functional programming. However, the use for these formulas for the proposed techniques in my thesis have to evaluated in the future.

7.3 Future Work

Beside the support in finding suitable styles, the development of a knowledge base on middleware-oriented architectural styles, technologies, design patterns, and other concepts can have additional benefits to the area of architecture and style development.

The knowledge base can be used during requirement engineering to search for technologies and suitable implementations of these technologies. In a feature driven approach these features could be used to formulate queries and the knowledge base can show suitable technologies implementing the requested features.

In conjunction with projects, like [WOP], the knowledge base can be used to support developers and software engineers in the prototyping and implementation phase of a project. The knowledge base allows comparisons of technologies in respect to certain criteria, which is a common use case in the early stages of a software project where the software stack has to be determined.

Another possible future use can come from such knowledge bases, when best practice knowledge on requirements and goals are stored in a knowledge base as well. When such knowledge would be available in a set of OWL knowledge bases, these knowledge bases could be combined, and in combination they could point to solutions for a given software problem by presenting suitable technology combinations. The software engineer would then evaluate the results and modify the problem description until the presented solution can become the basis of the desired software architecture.

A Analyze MidArch Styles

Bornhold did a case study on a regional trade information system [Bornhold 2006] with the MidArch method [Giesecke 2008]. In his work, he modeled middleware-oriented and generic architecture styles and used them for candidate architectures in his MidArch instance. Based on Bornholds work, in my chapter A, I analyze and model meta-information for these five styles based on the technique introduced chapter 4.

Section A.1 documents the modeling of the generic architectural style *Inversion of Control.* The middleware-oriented styles are documented in four sections. Section A.2 documents the *Spring Framework* and section A.3 documents the modeling for *Avalon Framework.* The *Cocoon* technology are represented by the more generic style *Cocoon Basic Style* and the specialized style *Cocoon with Flowscript.* The modeling of these two styles is documented in section A.4.

A.1 Inversion of Control

The architectural style *Inversion of Control* has already been discussed briefly in subsection 3.5.2. It is a generic architectural style and therefore not a MidArch style, however it plays the central role in [Bornhold 2006], because both framework induced styles (Spring and Avalon) are descendants of Inversion of Control. Furthermore [Fowler 2004] states that many frameworks implement the Inversion of Control pattern, which is the basis for the Inversion of Control style.

The *Inversion of Control* style is a generic and therefore not be technology induced style. It is documented and discussed in [Fowler 2004] and briefly described and formulated in ADL in [Bornhold 2006].

A.1.1 Analyze Documentation

The informal description in [Bornhold 2006] is in German therefore the determined clauses have been translated into English. The documentation of the style has no technical aspects, also it has no specified target domain or quality attributes. Therefore the knowledge aspects defined in section 4.3.1 cannot be used here. However the description of the Inversion of Control pattern [Fowler 2004] and style [Bornhold 2006] are very brief therefore only a view clauses are collected. The following list contains all clauses found in the documentation:

1. *Architectures* with *this style* have *containers* and *components*

2. *Components* are controlled by *a container*

3. Each *component* can only be controlled by *one container*
4. Each *component* is connected to *a container* by a *lifecycle connector*
5. There has to be at least *one container* in *an architecture with inversion of control*
6. *Service Locator* or *dependency injection* are different ways to realize *inversion of control*

A.1.2 Relating

The Inversion of Control style is a basis style therefore it is not related to other basis styles.

A.1.3 Codification

In this section the clauses get codified. Therefore the grammatical structure is analyzed. The noun phrases have already been marked in the enumeration above. Therefore the mapping of noun phrases on classes and individuals as well as the mapping of predicates and relationsships on properties is the next step.

- *This style* refers to *InversionOfControl* which is an instance of *ArchitectureFamily* (class)
- *InversionOfControl* hasComponentType *Container* and *Component*
- *Component* isControlledBy max one *Container* (rule)
- *LiveCycleConnector* connects *Component* and *Container*
- *InversionOfControl* has at least one *Container* (this is a constrain best expressed in an ADL)
- *InversionOfControl* hasInversionOfControlType *ServiceLocator* or *DependencyInjection*
- *ServiceLocator* or *DependencyInjection* form the concept *InversionOfControlType*

A.1.4 Formalization

```
Individual(InversionOfControl type(ArchitectureFamily)
   value(hasComponentType IoCComponent)
   value(hasComponentType IoCContainer)
   value(hasConnectorType IoCLifecycleConnector))

Individual(IoCComponent type(ComponentType)
   value(isEmbeddable true)
   value(hasPortType IoCLifecycleControlPort))
```

```
Individual(IoCContainer type(ComponentType)
   value(isContainer true)
   value(hasPortType IoCLifecycleControllerPort))

Individual(IoCLifecycleConnector type(ConnectorType)
   value(hasRoleType IoCLifecycleControllerRole)
   value(hasRoleType IoCLifecycleControlRole))

Individual(IoCLifecycleControllerPort type(PortType)
   value(acceptsRole IoCLifecycleControllerRole))

Individual(IoCLifecycleControlPort type(PortType)
   value(acceptsRole IoCLifecycleControlRole))

Individual(IoCLifecycleControllerRole type(RoleType))

Individual(IoCLifecycleControlRole type(RoleType))

EnumeratedClass(InversionOfControlType
   DependencyInjection
   ServiceLocator
)

ObjectProperty(hasInversionOfControlType
   domain(ArchitectureFamily) range(InversionOfControlType))

/* these definition have been introduced in chapter Model of the
   Knowledge Base and are shown here only for the sake of
   completeness */
Implies(
   Antecedent(
      hasComponentType(?style,?comp) isContainer(?comp,true)
      hasComponentType(?style,?embed) isEmbeddable(?embed,true)
      hasConnectorType(?style,?connector)
      hasPortType(?comp,?portS) acceptsRole(?portS,?roleS)
      hasPortType(?embed,?portC) acceptsRole(?portC,?roleC)
      hasRoleType(?connector,?roleS)
      hasRoleType(?connector,?roleC)
   )
   Consequent(controls(?comp,?embed))
)
DatatypeProperty(isContainer
   domain(ComponentType) range(xsd:boolean))
DatatypeProperty(isEmbeddable
   domain(ComponentType) range(xsd:boolean))
Implies(
   Antecedent(hasAncestor(?x,?y) isContainer(?y,true))
   Consequent(isContainer(?x,true))
)
```

```
Implies(
   Antecedent(hasAncestor(?x,?y) isEmbeddable(?y,true))
   Consequent(isEmbeddable(?x,true))
)
```

Listing A.1: Complete definition of the Inversion of Control style

A.2 Spring Framework

The *Spring Framework* is a Java framework for enterprise application. It is based on work from Johnson [2002]. It aims to be easier to use as J2EE or other Inversion of Control frameworks. As input source for the knowledge modeling, the following documents are used: Informal description of the *Spring Style* in [Bornhold 2006], the Spring website [Spring 2008] and the reference documentation of the Spring framework [Johnson et al. 2008].

A.2.1 Analyze Documentation

The modeling technique described in chapter 4 states that the first documentation to be analyzed is the informal description of an middleware-oriented style. Therefore the first document to be analyzed is [Bornhold 2006] followed by information from the website and finally the reference documentation.

Informal Description

The informal description in [Bornhold 2006] is written in German. However I translated the paraphrases into English. The result of the paraphrasing process is shown in the enumeration below. Style related noun phrases or objects (see section 2.1.3) are highlighted with *italics.*

1. *Spring* is an Inversion of Control based framework
2. *Spring* uses the *Dependency Injection* method to realize the Inversion of Control pattern
3. *Spring* is used with simpler components than Avalon
4. *Spring* components are *Plain old Java objects* (POJO)
5. *Spring* components have a *lifecycle*
6. *Container in Spring* is called *BeanFactory*
7. *Components* are called *Beans*
8. *Container* can automatically inject dependencies

9. *The container* uses *configuration files* to create and configure *components*
10. *An extended container type* of *BeanFactoy* is *ApplicationContext*
11. *ApplicationContext* supports *Internationalization*, platform independent access to resources, and *event handling* with an *Observer pattern*
12. *ApplicationContext* automatically instantiates *Singleton* objects
13. *Spring* supports configuration through *Setter Injection* and *Constructor Injection*
14. *Spring* supports the creation of components with *constructors* or through a *Factory Method*
15. *Beans* can receive information of their *lifecycle* through *Marker Interfaces*
16. *Spring* uses one *BeanFactory* or compatible container as central element, which contains all other components
17. *A container* is also *a component*

Web-Documents

The web-page of Spring [Spring 2008] contains, beside the reference documentation, a wiki, and an *about-page*. The reference documentation is analyzed below. The wiki contains many best practice information on implementation level which are not in the scope of this thesis. The *about-page*, however, contains a list of meta-information focusing on the basic idea behind *Spring* and main features of the framework. In the following a list of paraphrased sentences from the web-page is given categorized by the three knowledge areas.

Target Domain Aspect

18. *Spring* is the basis of *enterprise and business applications*

Technical Knowledge Aspect

19. *Spring* is *a Java-based technology*
20. *Spring* supports testing
21. *Spring* should work as *an integration platform*
22. *Spring* is a lightweight container which can work with loosely-coupled components
23. *Spring* is a common *abstraction layer* for *transaction management*
24. *Spring* has *a JDBC abstraction layer*

25. *Spring* supports *Data Access Objects* (DAO) with the technologies *Toplink*, *Hibernate*, *JDO*, and *iBATIS SQL Maps.*

26. *Spring* supports *Aspect Oriented Programming* (AOP)

27. *Spring* supports the *Model-View-Controller pattern* (MVC) through the technologies *JSP*, *Velocity*, *Tiles*, *iText*, and *POI.*

28. The *Spring middle tier* can be combined with a *web tier* based on other web MVC frameworks, like *Struts*, *WebWork*, or *Tapestry.*

29. *Spring* works with any *J2EE server* and most of it also in non-managed environments.

Quality Aspect

30. *Spring* is licensed under the terms of the Apache License.

Reference Documentation

The reference documentation [Johnson et al. 2008] is available in HTML and as a separate PDF. The documentation is for Spring version 2.5 and has an overview section which explains most features of the *Spring Framework* and the usage scenarios which are the basis for these features. The second chapter of this documentation discusses the changes since the previous version 2.0 of the Spring framework. The rest of the document is split in four parts which focus on the core technology, middle tier data access, the web, and the integration topic.

A detailed reading and analysis of the whole document would take too long, because the document comprises 587 pages. Therefore a selection of more important parts for the retrieval of meta-information is required. Because only general information on the parts and packages is required, it is enough to analyze the introductory sections of the documentation. Additionally these sections are read with the three knowledge aspects in mind. Below a list of paraphrased sentences from these introductory sections is given, subdivided by the three knowledge aspects.

Target Domain Aspect

31. *Spring* can be used for *web applications*

Technical Knowledge Aspect

32. The *Core package* provides the *Inversion of Control* and *Dependency Injection* features.

33. The *Context package* allows to access objects in a framework-style manner in a fashion somewhat reminiscent of a JNDI-registry

34. The *Context package* supports *interantionalization*, *event-propagation*, *resource-loading*, and *the transparent creation of contexts*

35. The *DAO package* provides a JDBC-abstraction layer and a *declarative transaction management*

36. The *ORM package* integrates *object-rational mapping APIs*, like *JPA*, *JDO*, *Hibernate*, *Oracle TopLink*, and *iBATIS SQL Maps.*

37. The *AOP package* provides an *AOP Alliance*-compliant *aspect programming implementation*

38. The *AOP package* provides source-level metadata functionality for behavioral information similar to .NET attributes.

39. The *AOP package* supports *method-interceptors* and *pointcuts* to decouple code implementing functionality

40. The *Web package* provides basic web-oriented integration features: multipart file-upload, initialization of the IoC container, and a web-oriented application context.

41. The *Web package* allows the integration of *Struts* and *WebWork*

42. The *MVC package* provides a *MVC implementation* for *web-applications*

43. The *MVC package* allows the use of different *view technologies*, like *JSP*, *JSTL*, *Tiles*, *Velocity*, *FreeMaker*, *XSLT*, *JasperReports* or *diverse document views*

44. *Spring* can be used for *web applications* running in *a servlet container*

45. *AOP* is used to provide *declarative enterprise services*, as *a replacement for EJB declarative services*

46. *Spring's AOP* can use a *Schema-based AOP* support or *AspectJ*

47. *Schema-based AOP* does not depend on *Java 5*

48. *AspectJ* is an *AOP* technology

49. *Spring* fosters the use of unit testing with packages like *JUnit* or *TestNG*

50. *Spring* provides *mock objects*

51. *Spring* supports *integration testing*

52. *Spring* provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO

53. *Spring* supports *delarative transaction management*

54. *Spring* is a *multi-tier* platform

55. *The Spring Framework* can run in servlet and portlet environments

56. *Spring Web Flow* integrates with *Spring MVC*, *Struts*, and *JSF*

57. *Spring* integrates *web frameworks* such as *JSF*, *Struts*, *WebWork*, and *Tapestry.*

58. *Spring* supports *the portlet standard JSR-168*

59. *Spring* supports *remoting technologies Remote Method Invocation (RMI)*, *Spring's HTTP invoker*, *Hessian*, *Burlap*, *JAX-RPC*, *JAX-WS*, and *Java Message Service (JMS).*

60. *Spring* works with *EJB*

61. *Spring* helps with *stateless session beans*

62. *Spring* supports *Common Client Interface* of *the Java Connector Architecture (JCA)*

63. *The Java Connector Architecture* is a *J2EE standard* for *the access* to *enterprise information systems*

64. *JCA* defines *a service provider interface (SPI)* and *a common client interface*

65. *Spring* integrates with *email services*

66. *Spring* supports *scheduling* and *thread pooling* with *JDK 1.3 Timer* and *Quartz*

Quality Aspect

67. *Spring* supports a simpler API for *programmatic transaction management* compared to *JTA*

68. *Spring* is *an integration platform*

69. *Spring* is widely used and has support by *a large community* and several *software companies*

A.2.2 Relating

The informal modeling task has produced a long list of paraphrased sentences. They contain a large number of terms and expressions, which have or express relationships. The center of the analysis is the *Spring Framework* which is associated with the *Spring Style* from [Bornhold 2006]. The style is a specialization of the *Inversion of Control* style.

The list of sentences mentions many different technologies and pattern, but the knowledge base does not contain any defined technologies. However *Spring* is a Java-based technology and programming languages are already present in the knowledge base. Furthermore the knowledge base has information about test technologies, like *JUnit* which are also used in *Spring*.

Based on this analysis the following list of relationships can be formulated:

1. *Spring* is a sub-class of *Inversion of Control*
2. *Spring* is a *Java-based* technology
3. *Spring* supports the test method *JUnit* (it also supports *TestNG* which is not known by the knowledge base at this point)

A.2.3 Codification

The sentences and clauses collected in the document analysis task define many different properties of the *Spring* style. To keep track of the mapping of properties to the codification description, I prefix the codification with the corresponding enumeration.

1 *Spring* hasParent *InversionOfControl*

2,13 *Spring* hasDependencyInjectionType *SetterInjection* and *ConstructorInjection*

2 *Spring* hasInversionOfControlType *DependencyInjection*

3 *Spring* hasComponentComplexity *low*

4,7 *Bean* are *POJO*

5,7 *Bean* hasParent *IoCComponent*

6 *BeanFactory* is class *ComponentType* and hasParent *IoCContainer*

8 *BeanFactory* performsDependencyInjection *yes*

9 *BeanFactory* createsComponents *yes* configuresComponents *yes*

10 *ApplicationContext* hasParent *BeanFactory*

11 *ApplicationContext* supportsTechnology *Internationalization*

11 *ApplicationContext* supportsPattern *ObserverPattern*

11 *ApplicationContext* supportsTechnology *PlatformIndependentResourceAccess*

12 *ApplicationContext* supportsPattern *Singleton*

14 *Spring* supportsComponentConstructionMethod *ComponentConstructionMethod*

14 *Constructor* and *FactoryMethod* instance of *ComponentConstructionMethod*

15 *Bean* hasPortType *MarkerInterface*

15 *MarkerInterface* hasParent *IoCLifecycleControlPort*

18,31 *Spring* supportsApplicationType *ApplicationType* (class) with values *Enterprise-Application*, *BusinessApplication*, and *WebApplication*

19 *Spring* supportsProgrammingLanguage *Java* and *Csharp*

21 *Spring* providesFeature *Integration*

22 *BeanFactory* hasContainerType *LightWeightContainer*

25 *DAO* sameAs *DirectAccessObject*

26 *AspectOrientedProgramming* sameAs *AOP*

28,41,57 *Spring* hasPackage *SpringMVCPackage*

28,41,57 *SpringMVCPackage* supportsTechnology *Struts*, *WebWork*, *JSF*, and *Tapestry*

28 *Struts*, *WebWork*, and *Tapestry* supportPattern *MVC*

29 *Spring* supportsTechnology *J2EEServer*

30 *Spring* hasLicense *ApacheLicense*

30 *ApacheLicense* instance of *License*

30 *ApacheLicense* isType *OpenSourceLicense*

32–43 *Spring* hasPackage *Package* with values *SpringCorePackage*, *SpringContextPackage*, *SpringDAOPackage*, *SpringORMPackage*, *SpringAOPPackage*, and *Spring-MVCPackage*

32–43 *Package* (class) has restrictions: supportsPattern, supportsTechnology, provides-Technology

32 *SpringCorePackage* providesTechnology *Spring*

33 *SpringContextPackage* supportsTechnology *JNDI*

34 *SpringContextPackage* supportsTechnology *Internationalization*

34 *SpringContextPackage* providesFeature *EventPropagation*, *ResourceLoading*, *TransparentContextCreation*

23,24,35,53 *SpringDAOPackage* supportsTechnology *JDBC* and *TransactionManagement* (class)

35 *TransactionManagement* (class) has individuals *DeclarativeTransactionManagement* and *ProgrammaticTransactionManagement*

25,36 *SpringORMPackage* supportsTechnology *Toplink*, *Hibernate*, *iBATIS SQL Maps*, *JDO*, and *JPA*

25,36,52 *Toplink*, *Hibernate*, *JDO*, *JPA*, and *iBATIS SQL Maps* supportsPattern *DirectAccessObject* is instance of *DesignPattern*

26,37 *SpringAOPPackage* supportsPattern *AspectOrientedProgramming*

37 *SpringAOPPackage* conformsTo *AOPAllianceStandard*

38 *SpringAOPPackage* providesFeature *BehavioralInformation*

39 *SpringAOPPackage* providesFeature *MethodInterceptors* and *Pointcuts*

39 *AOPFeature* (class) has parent *Feature* has instances *MethodInterceptors* and *Pointcuts*

40 *SpringMVCPackage* providesComponent *WebApplicationContext*

40 *WebApplicationContext* hasParent *ApplicationContext*

40 *SpringMVCPackage* providesFeature *MultipartFileUpload*

27,42,43 *SpringMVCPackage* supportsTechnology *JSP*, *Velocity*, *Tiles*, *iText*, and *POI*

27,43 *JSP*, *Velocity*, *Tiles*, *iText*, and *POI* supportsPattern *ModelViewController*

27 *ModelViewController* sameAs *MVC*

44,55 *Spring* supportsTechnology *ServletContainer*

45 *SpringAOPPackage* providesFeature *SpringDeclarativeEnterpriseService*

45 *SpringDeclerativeEnterpriseService* and *EJBDeclarativeService* are instance of *DeclerativeService* (class)

46 *SpringAOPPackage* supportsTechnology *SchemaBasedAOP* and *AspectJ*

47 *AspectJ* requires *Java5*

47 *Java5* hasParent *Java* instance of *ProgrammingLanguage* (class)

48 *AspectJ* and *SchemaBasedAOP* instance of *AOPTechnology*

20,49 *Spring* supportsTestMethod *UnitTest* and *Stubs*

49 *UnitTest* hasTechnology *JUnit* and *TestNG*

50 *Stubs* hasTechnology *MockObject*

51 *Spring* supportsTestMethod *IntegrationTesting*

54 *Spring* instance of *MultiTierPlatform*

55,58 *Spring* supportsTechnology *JSR168Container* instance of *Portlet*

56 *SpringWebFlow* integratesWith *SpringMVCPackage*, *Struts*, *JSF*

59 *Spring* supportsTechnology *RMI*, *SpringHTTPinvoker*, *Hessian*, *Burlap*, *JAX-RPC*, *JAX-WS*, and *JMS*

59 *RMI*, *SpringHTTPinvoker*, *Hessian*, *Burlap*, *JAX-RPC*, *JAX-WS*, and *JMS* are instance of *RemotingTechnologies*

60 *Spring* worksWith *EJB*

61 *Spring* providesFeature *StatelessSessionBean*

62 *Spring* supportsTechnology *CCI*

62 *CCI* isPartOf *JCA*

62 *CCI*, *SPI*, *JCA* instance of *Technology*

64 *SPI* isPartOf *JCA*

65 *Spring* supportsTechnology *EMailService*

66 *Spring* supportsTechnology *JDKTimer* and *Quartz*

66 *JDKTimer* and *Quartz* providesFeature *Scheduling*, *ThreadPooling*

68 *Spring* instance of *IntegrationPlatform*

69 *Spring* isSupported *excellent*

A.2.4 Formalization

```
ObjectProperty(worksWith Symmetric
   domain(unionOf(Technology ProgrammingLanguage))
   range(unionOf(Technology ProgrammingLanguage)))

DatatypeProperty(hasComponentComplexity Functional
   domain(ArchitectureFamily) range(oneOf(
      "high"
      "low"
)))

DatatypeProperty(isSupported Functional
   domain(MiddlewarOrientedArchitectureFamily) range(oneOf(
      "excellent"
      "good"
      "satisfactory"
      "adequate"
      "inadequate"
      "missing"
)))

EnumeratedClass(ApplicationType
   EnterpriseApplication
   BusinessApplication
   WebApplication
)

ObjectProperty(providesFeature
   domain(unionOf(
     ArchitectureFamily
     ComponentType
     ConnectorType
     Package
     Technology)))
   range(Feature)
)

ObjectProperty(supportsApplicationType
   domain(ArchitectureFamily) range(ApplicationType))

Individual(ModelViewController type(DesignPattern))
Individual(MVC sameAs(ModelViewController))

Class(Feature partial owl:Thing)

Individual(Integraion type(Feature))
Individual(EventPropagation type(Feature))
Individual(ResourceLoading type(Feature))
Individual(TransparentContextCreation type(Feature))
Individual(BehvarioralInformation type(Feature))
Individual(MultipartFileUpload type(Feature))
Individual(StatelessSessionBean type(Feature))

Class(DeclarativeService partial Feature)

Individual(SpringDeclarativeEnterpriseService
   type(DeclerativeService))
Individual(EJBDeclarativeService type(DeclerativeService))

Class(AOPFeature partial Feature)
Individual(MethodInterceptors type(AOPFeature))
Individual(Pointcuts type(AOPFeature))

Individual(JDKTimer type(Technology)
   value(providesFeature Scheduling)
   value(providesFeature ThreadPooling)
)
Individual(Quartz type(Technology)
   value(providesFeature Scheduling)
   value(providesFeature ThreadPooling)
)

Individual(EMailService type(Technology))
Individual(EJB type(Technology))

ObjectProperty(isPartOf domain(Technology) range(Technology))

Individual(JCA type(Technology))
Individual(CCI type(Technology)
   value(isPartOf JCA)
)
Individual(SPI type(Technology)
   value(isPartOf JCA)
)

Class(RemotingTechnology partial Technology)
Individual(RMI type(RemotingTechnology))
Individual(SpringHTTPinvoker type(RemotingTechnology))
Individual(Hessian type(RemotingTechnology))
Individual(Burlap type(RemotingTechnology))
Individual(JAX–RPC type(RemotingTechnology))
Individual(JAX–WS type(RemotingTechnology))
Individual(JMS type(RemotingTechnology))

Individual(IntegrationTesting type(TestConcept))
Individual(Stubs type(TestConcept)
   isImplementedBy(MockObject)
)
Individual(UnitTest type(TestConcept)
   isImplementedBy(JUnit)
   isImplementedBy(TestNG)
)

Class(License partial owl:Thing)
Class(OpenSourceLicense partial License)
Individual(GPL type(OpenSourceLicense))
Individual(ApacheLicense type(OpenSourceLicense))
Individual(BSD type(OpenSourceLicense))
Individual(LGPL type(OpenSourceLicense))
Individual(MozillaLicense type(OpenSourceLicense))

ObjectProperty(hasLicense domain(Technology) range(License))

Individual(Csharp type(ProgrammingLanguage))
Individual(SpringBean type(ComponentType)
   value(hasParent IoCComponent)
   value(hasPortType MarkerInterface)
)

Individual(MarkerInterface type(PortType)
   value(hasParent IoCLifecycleControlPort)
)

DatatypeProperty(hasContainerType Functional
   domain(ComponentType) range(oneOf(
   "LightWeightContainer"
   "BulkContainer"
)))

DatatypeProperty(performsDependencyInjection Functional
   domain(ComponentType) range(xsd:boolean))

DatatypeProperty(createComponents Functional
   domain(ComponentType) range(xsd:boolean))

DatatypeProperty(configuresComponents Functional
   domain(ComponentType) range(xsd:boolean))

Individual(SpringBeanFactory type(ComponentType)
   value(hasParent IoCContainer)
   value(performsDependencyInjection yes)
   value(createComponents yes)
   value(configuresComponents yes)
   value(hasContainerType LightWeightContainer)
)

ObjectProperty(supportsTechnology
   domain(unionOf(
      ComponentType
      ArchitectureFamily
      ConnectorType
      Package))
   range(Technology))

Individual(Internationalization type(Technology))
Individual(PlatformIndependenResourceAccess type(Technology))

Individual(ObserverPattern type(DesignPattern))
Individual(Singleton type(DesignPattern))

Individual(SpringApplicationContext type(ComponentType)
   value(hasParent SpringBeanFactory)
   value(supportsTechnology Internationalization)
   value(supportsPattern ObserverPattern)
   value(supportsTechnology PlatformIndependenResourceAccess)
   value(supportsPattern Singleton)
)

Individual(SpringWebApplicationContext type(ComponentType)
   value(hasParent SpringApplicationContext)
)

Class(ComponentConstructionMethod partial DesignPattern)

Individual(Constructor type(ComponentConstructionMethod))
Individual(FactoryMethod type(ComponentConstructionMethod))

Individual(DirectAccessObject type(DesignPattern))
Individual(DAO sameAs(DirectAccessObject))
Individual(AspectOrientedProgramming type(DesignPattern))
Individual(AOP sameAs(AspectOrientedProgramming))

Individual(Struts type(Technology))
Individual(WebWork type(Technology))
Individual(JSF type(Technology))
Individual(Tapestry type(Technology))
Individual(J2EEServer type(Technology))
Individual(JDBC type(Technology))
Individual(Toplink type(Technology)
   value(supportsPattern DAO)
)
Individual(Hibernate type(Technology)
   value(supportsPattern DAO)
)
Individual(JDO type(Technology)
   value(supportsPattern DAO)
)
Individual(JPA type(Technology)
   value(supportsPattern DAO)
)
Individual(iBatisSQLMaps type(Technology)
   value(supportsPattern DAO)
)
Individual(JSP type(Technology)
   value(supportsPattern ModelViewController)
)
Individual(Velocity type(Technology)
   value(supportsPattern ModelViewController)
)
Individual(Tiles type(Technology)
   value(supportsPattern ModelViewController)
```

```
)
Individual(iText type(Technology)
   value(supportsPattern ModelViewController)
)
Individual(POI type(Technology)
   value(supportsPattern ModelViewController)
)
Individual(SpringWebFlow type(Technology)
   value(worksWith SpringMVCPackage)
   value(worksWith Struts)
   value(worksWith JSF)
)

Class(Servlet parital Technology)
Individual(ServletContainer type(Servlet)
   value(supportsProgrammingLanguage Java)
)

Class(Portlet partial Technology)
Individual(JSR168Container type(Portlet)
   value(supportsProgrammingLanguage Java)
)

Class(TransactionManagement partial Technology)
Individual(DeclarativeTransactionManagement
   type(TransactionManagement))
Individual(ProgrammaticTransactionManagement
   type(TransactionManagement))

ObjectProperty(providesTechnology domain(Package) range(Technology))

Class(Package partial owl:Thing
   restriction(hasLicense someValueFrom(License))
   restriction(supportsPattern someValueFrom(Pattern))
   restriction(supportsTechnology someValuesFrom(Technology))
   restriction(providesTechnology someValuesFrom(Technology))
)

Class(Standard partial owl:Thing)

Individual(AOPAllianceStandard type(Standard))

ObjectProperty(providesComponent
   domain(Package) range(ComponentType))
ObjectProperty(conformsTo
   domain(Technology Package) range(Standard))

Individual(SpringMVCPackage type(Package)
   value(supportsTechnology Struts)
   value(supportsTechnology WebWork)
   value(supportsTechnology JSF)
   value(supportsTechnology Tapestry)
   value(providesComponent SpringWebApplicationContext)
   value(providesFeature MultipartFileUpload)
   value(supportsTechnology JSP)
   value(supportsTechnology Velocity)
   value(supportsTechnology Tiles)
   value(supportsTechnology iText)
   value(supportsTechnology POI)
)

Individual(SpringCorePackage type(Package)
   value(providesTechnology Spring)
)

Individual(SpringDAOPackage type(Package)
   value(supportsTechnology JDBC)
   value(supportsTechnology DeclarativeTransactionManagement)
   value(supportsTechnology ProgrammaticTransactionManagement)
)

Individual(SpringContextPackage type(Package)
   value(supportsTechnology Internationalization)
   value(providesFeature EventPropagation)
   value(providesFeature ResourceLoading)
   value(providesFeature TransparentContextCreation)
)

ObjectProperty(requires domain(Technology) range())

Individual(Java5 type(ProgrammingLanguage)
   value(hasParent Java))

Class(AOPTechnology partial Technology)

Individual(AspectJ type(AOPTechnology)
   value(requires Java5)
)
Individual(SchemaBasedAOP type(AOPTechnology))

Individual(SpringAOPPackage type(Package)
   value(supportsPattern AspectOrientedProgramming)
   value(conformsTo AOPAllianceStandard)
   value(providesFeature BehavioralInformation)
   value(providesFeature MethodInterceptors)
   value(providesFeature Pointcuts)
   value(providesFeature SpringDeclarativeEnterpriseService)
   value(supportsTechnology SchemaBasedAOP)
   value(supportsTechnology AspectJ)
)

Individual(SpringORMPackage type(Package)
   value(supportsTechnology Toplink)
   value(supportsTechnology Hibernate)
   value(supportsTechnology JDO)
   value(supportsTechnology JPA)
   value(supportsTechnology iBatisSQLMaps)
)

Class(MultiTierPlatform partial
   MiddlewareOrientedArchitecturePlatform)
Class(IntegrationPlatform partial
   MiddlewareOrientedArchitecturePlatform)

Individual(Spring
   type(intersectionOf(MiddleTierPlatform IntegrationPlatform))
   value(hasParent InversionOfControl)
   value(hasDependencyInjectionType SetterInjection)
   value(hasDependencyInjectionType ConstructorInjection)
   value(hasInversionOfControlType DependencyInjection)
   value(hasComponentComplexity low)
   value(hasComponentType SpringBean)
   value(hasComponentType SpringApplicationContext)
   value(hasComponentType SpringBeanFactory)
   value(supportsComponentConstructionMethod Constructor)
   value(supportsComponentConstructionMethod FactoryMethod)
   value(supportsApplicationType EnterpriseApplication)
   value(supportsApplicationType BusinessApplication)
   value(supportsApplicationType WebApplication)
   value(supportsProgrammingLanguage Java)
   value(supportsProgrammingLanguage Csharp)
   value(providesFeature Integration)
   value(hasPackage SpringMVCPackage)
   value(hasPackage SpringCorePackage)
   value(hasPackage SpringContextPackage)
   value(hasPackage SpringDAOPackage)
   value(hasPackage SpringORMPackage)
   value(hasPackage SpringAOPPackage)
   value(supportsTechnology J2EEServer)
   value(supportsTechnology ServletContainer)
   value(supportsTechnology JSR168Container)
   value(supportsTechnology RMI)
   value(supportsTecdochnology SpringHTTPinvoker)
   value(supportsTechnology Hessian)
   value(supportsTechnology Burlap)
   value(supportsTechnology JAX-RPC)
   value(supportsTechnology JAX-WS)
```

```
    value(supportsTechnology JMS)
    value(supportsTechnology CCI)
    value(supportsTechnology EMailService)
    value(supportsTechnology JDKTimer)
    value(supportsTechnology Quartz)
    value(supportsTestMethod UnitTest)
    value(supportsTestMethod Stubs)
    value(supportsTestMethod IntegrationTesting)
    value(worksWith EJB)
    value(providesFeature StatelessSessionBean)
    value(isSupported excellent)
)
```

Listing A.2: Complete definition of the Spring Architecture Family and all relevant new concepts, individuals, and properties required for it

A.3 Avalon Framework

The *Apache Avalon Project* [Avalon 2004] developed an Inversion of Control framework, called *Avalon Framework*. Since 2004 the project is closed and the Avalon Framework is now part of the *Excalibur Project*. In [Bornhold 2006] the framework and its corresponding style is called *Avalon*, therefore I use the name *Avalon* for the knowledge modeling as well.

Beside the style description from [Bornhold 2006], the sections *Fortess* and *Framework* from the web-page of *Apache Excalibur* [Excalibur 2007] are processed.

A.3.1 Analyze Documentation

The first document to be analyzed is the informal description from [Bornhold 2006]. The resulting phrases are cataloged in section A.3.1. The results from the paraphrasing of Avalon web-page is cataloged in section A.3.1.

Informal Description

The informal description in [Bornhold 2006] is written in German. However I translated the paraphrases into English. The result of the paraphrasing process is shown in the enumeration below. Style related noun phrases or objects (see section 2.1.3) are highlighted with *italics*.

1. *Avalon* has always one *container* to control *the components*
2. *A container* is *a component* which can be controlled by another *container*
3. *Avalon* allows to build a hierarchy of *containers*
4. *Components* access each other over *roles*

5. *Each role* has its own *interface*
6. *The service manager* controls *the access* to other *components*
7. *The service manager* is *a component type*
8. *A component selector* handles different *implementations of a role*
9. *The Avalon container* is *the container component* of *Avalon*
10. *The component selector* is *a component*

Technical Documentation

The technical documentation of the *Avalon Framework* is compiled on the web-page of the Excalibur project. Compared to the *Spring Framework* the documentation has not reached the same level of quality. It is superficial and does not provide introductory information. However the web-page has two sections named *Fortess* and *Framework* which contain relevant information for the three knowledge aspects defined in section 4.3.1. Below a list of paraphrased sentences from these sections is given, subdivided by the three knowledge aspects.

Target Domain Aspect

11. *Avalon* can be used for *servlet* and *Swing application*

Technical Knowledge Aspect

12. *A Role* is represented by *an interface*
13. The concerns supported are configuration, external, component use, management, role, and execution.
14. The concerns are represented by interfaces
15. The component serviceable is a container
16. The service manager is used to get component instances.
17. The service manage does the service lookup
18. If a role has more than one implementation than a service selector can be used to choose between these implementations
19. *A service selector* can work with hashtable or dynamic lookup
20. *The lifecycle package* contains classes supporting portable extensions
21. *Datasource package* handles pooled connections or uses J2EE server's data source management

22. *Monitor package* supports *active* and *passive* resource monitoring
23. *Pool package* handles resource pooling
24. *Sourceresolver* is a component used to find the source by an URI
25. *Store package* supports storing named object in memory or on file system
26. *XMLUtil package* provides XML utility functions
27. *Avalon* supports Swing-based applications
28. *Avalon* supports Servlet-based applications
29. *Avalon* can bind to JNDI
30. *JNDI* provides resource management
31. *Avalon* provides *a container component*
32. *Avalon* supports pattern *seperation of concern*
33. *Avalon* supports pattern *aspect oriented programming* through interfaces or AspectJ
34. *Avalon* supports quality of service, authentication, authorization

Quality Aspect

35. *Avalon project* replaced by the *Excalibur project*
36. *Documentation* is out of date and incomplete

A.3.2 Relating

The informal modeling task has produced a list of paraphrased sentences which contain numerous terms and expressions. These terms have or express relationships to other terms in the knowledge base. In this relating task relationships to existing individuals in the knowledge base are determined. The following list catalogs all determined relationships.

37. *Avalon* is a *Inversion of Control* style
38. *Avalon* supports technology *JNDI*
39. *Avalon* supports pattern *AOP*
40. *Avalon* supports pattern *SOC*
41. *Avalon* is a *Java-based* technology

A.3.3 Codification

1,9,31 *Avalon* hasComponentType *AvalonContainer* and *AvalonComponent*

2,3 *AvalonContainer* isEmbeddable *true*

2,3 *AvalonContainer* isContainer *true*

4,5,12 *Role* sameAs *Interface* sameAs *PortType* (The term *Role* is not defined to minimize interpretation errors, because the term *Role* is close to *RoleType*, but describes a complete different concept.)

6 *AvalonServiceManager* providesAccessControlTo *AvalonComponent*

7 *AvalonServiceManager* instance of *ComponentType*

8,18,19 *AvalonServiceSelector* provideFeature *RoleSelection*

10 *Avalon* hasComponentType *AvalonServiceSelector*

11 *Avalon* supportsTechnology *ServletContainer*, *Swing*

14 *Concern* sameAs *Interface*

14 *Configuration*, *ExternalComponentUse*, *Management*, *Execution* instance of *Concern*

15 *Avalon* hasComponentType *Serviceable* isContainer *true*

16,17 *AvalonServiceManager* providesFeature *ComponentLookup*

20 *AvalonLifecyclePackage* providesFeature *Lifecycle*

21 *Avalon* hasPackage *AvalonDatasourcePackage*

21 *AvalonDatasourcePackage* providesTechnology *J2EEDatasourceManagement*, *AvalonDatasourceManagement*

21 *J2EEDatasourceManagement*, *AvalonDatasourceManagement* providesFeature *DatasourceManagement*

22 *AvalonMonitorPackage* providesTechnology *ActiveResourceMonitoring*, *PassiveResourceMonitoring* instance of *ResourceMonitoring* partial *Technology*

23 *AvalonPoolPackage* providesFeature *ResourcePooling*

24 *Avalon* hasComponentType *AvalonSourceresolver*

24 *AvalonSourceresolver* providesFeature *SourceLookup*

25 *Avalon* hasPackage *AvalonStorePackage* providesFeature *DataStorage*

26 *Avalon* hasPackage *XMLUtilPackage* providesFeature *XMLProcessing*

27 *Avalon* supportsTechnology *Swing*

28 *Avalon* supportsTechnology *ServletContainer*

29,38 *Avalon* supportsTechnology *JNDI*

30 *JNDI* providesFeature *ResourceManagement*

32,40 *Avalon* supportsPattern *SeparationOfConcern* sameAs *SOC*

33,39 *Avalon* supportsTechnology *AspectJ* supportsPattern *AOP*

34 *Avalon* supportsTechnology *Authentication*, *Authorization*

35,36 *Avalon* documentationQuality *low*

37 *Avalon* hasParent *InversionOfControl*

41 *Avalon* supportsProgrammingLanguage *Java*

A.3.4 Formalization

```
Class(Interface sameAs PortType)

Individual(AvalonServiceManager type(ComponentType)
   value(providesFeature ServiceLocator)
)

Individual(AvalonContainer type(ComponentType)
   value(hasParent IoCContainer)
   value(isEmbeddable true)
)
Individual(AvalonServiceable sameAs AvalonContainer)
)

Individual(AvalonComponent type(ComponentType)
   value(hasParent IoCComponent)
)

Individual(RoleSelection type(Feature))

Individual(AvalonServiceSelector type(ComponentType)
   value(providesFeature RoleSelection)
)

Individual(Swing type(Technology))

Individual(Lifecycle type(Feature))

Individual(AvalonLifecyclePackage type(Package)
   value(providesFeature Lifecycle)
)

Individual(AvalonDatasourceManagement type(Technology)
   value(providesFeature DatasourceManagement)
)
Individual(J2EEDatasourceManagement type(Technology)
   value(providesFeature DatasourceManagement)
)

Individual(AvalonDatasourcePackage type(Package)
   value(providesTechnology AvalonDatasourceManagement)
   value(providesTechnology J2EEDatasourceManagement)
)

Class(ResourceMonitoring parital type(Technology))

Individual(ActiveResourceMonitoring type(ResourceMonitoring))
Individual(PassiveResourceMonitoring type(ResourceMonitoring))

Individual(AvalonMonitorPackage type(Package)
   value(providesTechnology ActiveResourceMonitoring)
   value(providesTechnology PassiveResourceMonitoring)
)

Individual(ResourcePooling type(Feature))

Individual(AvalonSourceResolver type(ComponentType)
   value(providesFeature SourceLookup)
)

Individual(AvalonPoolPackage type(Package)
   value(providesFeature ResourcePooling)
   value(providesComponent type(AvalonSourceResolver))
)

Individual(DataStorage type(Feature))

Individual(AvalonStorePackage type(Package)
   value(providesFeature DataStorage)
)

Individual(XMlProcessing type(Feature))

Individual(AvalonXMLUtilPackage type(Package)
   value(providesFeature XMLProcessing)
)

Individual(JNDI type(Technology)
  value(providesFeature(ResourceManagement))
)

Individual(SeparationOfConcern type(DesignPattern))
Individual(SOC sameAs SeparationOfConcern)

Individual(Authentication type(Technology))
Individual(Authorization type(Technology))

Individual(AvalonLifecycleConnector type(ConnectorType)
  value(hasParent IoCLifecycleConnector)
)

Individual(Avalon type(MiddlewareOrientedArchitectureFamily)
  value(hasParent InversionOfControl)
  value(hasConnectorType AvalonLifecycleConnector)
  value(hasComponentType AvalonServiceSelector)
  value(hasComponentType AvalonSourceResolver)
  value(hasComponentType AvalonServiceManager)
  value(hasComponentType AvalonContainer)
  value(hasComponentType AvalonComponent)
  value(supportsTechnology ServletContainer)
  value(supportsTechnology Swing)
  value(supportsTechnology JNDI)
  value(hasPackage AvalonLifecyclePackage)
  value(hasPackage AvalonDatasourcePackage)
  value(hasPackage AvalonMonitorPackage)
  value(hasPackage AvalonStorePackage)
  value(supportsPattern SOC)
  value(supportsTechnology AspectJ)
  value(supportsTechnology Authentication)
  value(supportsTechnology Authorization)
  value(documentationQuality low)
  value(supportsProgrammingLanguage Java)
)

Implies(
  Antecedent(hasAncestor(?ancestor,?component)
     ComponentType(?ancestor) ComponentType(?component)
     isContainer(?ancestor, true))
  Consequent(isContainer(?component, true))
)
Implies(
  Antecedent(hasAncestor(?ancestor,?component)
     ComponentType(?ancestor) ComponentType(?component)
     isEmbeddable(?ancestor, true))
  Consequent(isEmbeddabe(?component, true))
)
```

Listing A.3: Complete definition of the Avalon Architecture Family and all relevant new concepts, individuals, and properties required for it

A.3.5 Reviewing

The new *Avalon Architecture Family* introduced new technologies, features, and pattern, as well as other concepts, which may also be present in the Spring Framework. Therefore I analyze in the reviewing task the Spring Framework to find matches with the new concepts. Also I check for new rules which enrich the reasoning and ease the querying.

1. *Hibernate*, *iBatisSQLMaps*, *JDO*, *JPA*, *Toplink* providesFeature *DatasourceManagement*
2. *Spring* supportsTechnology *Swing*
3. If *a package* supports *a technology* then *the architectural family* owning *the package* also supports *this technology*
4. If *a package* supports *a pattern* then *the architectural family* owning *the package* also supports *this pattern*
5. If *a technology* supports *a pattern* then *the supporter of the technology* also supports *the pattern*
6. If *a technology*, *a package*, or *a component* provides *a feature* then *the architectural style* for *this technology*, *package*, or *component* provides *the same feature*
7. *Hibernate*, *iBatisSQLMaps*, *JDO*, *JPA*, *Toplink* providesFeature *DataStorage*
8. *SpringContextPackage* providesFeature *Lifecycle*
9. *Spring* providesFeature *ResourcePooling*
10. *Spring* providesFeature *XMLProcessing*

A.4 Cocoon Styles

The two *Cocoon* styles described in [Bornhold 2006] are very similar. The difference is that the first style called *Cocoon Basic Style* does not use *FlowScript*, but the second style, called *Cocoon with FlowScript* does. Therefore it is an extension of the basic style.

Because of this close relationship of these two styles, I perform the analysis and formalization for them together. The input documents are the Cocoon web-site [Cocoon 2008] and the style description from [Bornhold 2006].

A.4.1 Analyze Documentation

The analysis starts with the *informal style description* from [Bornhold 2006]. Then a selection of pages from the web-site are analyzed to get more detailed information.

Informal Description

1. *Apache Cocoon* is a framework to create *web application*
2. *Cocoon* uses the *Pipe-and-Filter* desgin pattern
3. *The processing* of *a request* starts at *the sitemap*
4. *The matcher* selects a suitable *pipeline* for a given *request*
5. *A selector* is a more complex *matcher*
6. *An action*, *a selector*, and *a matcher* support the selection of *a pipeline* for a given *request*
7. *An action* can manipulate *runtime parameter* of *a pipeline*
8. *A pipline* is *a sequence of pipes and filters*
9. *a filter* processes *SAX-Events*
10. *a pipe* relays *SAX-Events*
11. *A generator* creates *SAX-Events*
12. *A transformator* is *a filter*
13. *A serializer* only reads *SAX-Events*
14. *A pipeline* starts with *a generator*, *an aggregator* or *a reader*

Technical Documentation

The *Cocoon* web-site comprises of documentation for various versions of *Cocoon*. The two more recent versions are 2.1 and 2.2 which are the basis for the analysis. The major change in *Cocoon* from 2.1 to 2.2 is the shift from *Avalon* to *Spring* as the underlying technology. To ease the migration from *Avalon* to *Spring*, *Cocoon* provides a bridge which allows the use of *Avalon* components in a *Spring* container.

The technical documentation for *Cocoon* is quite comprehensive, therefore a preselection of pages and document parts has to be made. Unlike the *Spring* documentation, the *Cocoon* documentation does not have an overview section to each subject. Therefore some information can only be gathered by examining specification documents.

All selected documents, parts, and sections have been analyzed with the three knowledge aspects in mind. The results are described in the three corresponding sections below.

Target Domain Aspect

15. *Cocoon* can be used for *web portals*
16. *Cocoon* supports multi-channel web publishing
17. *Cocoon* supports static content generation

Technical Knowledge Aspect

18. *Cocoon* has *a authentication framework*
19. *The authentication framework* provides *module management* which allows component-based configuration
20. *The authentication framework* provides *application management* which allows to configure different applications on the same engine
21. *The authentication framework* provides *user management*
22. *The user management* supports *a role and user based authentication model*
23. *Cocoon* provides *a session management*
24. *Cocoon* requires *Java*
25. *Cocoon* requires *a servlet engine* such as *Jetty* or *Apache Tomcat*
26. *Cocoon* 2.2 is *Spring-based*
27. *Cocoon* 2.1 is *Avalon-based*
28. *A cocoon block* is a unit of modularization comparable to *Eclipse plugins* or *OSGi bundles* (or *Package* in the knowledge base of this thesis)
29. *Cocoon* provides general servlet services
30. *Cocoon* provides special services in form of pipelines
31. *Cocoon* provides component services for *Spring beans* and *Avalon services*
32. *Cocoon* provides *flow control*
33. *Cocoon* provides *internationalization*
34. *Cocoon* supports portlet technology *JSR-168 container*
35. *Cocoon* can process *XML input* from *files*, *services*, and *databases*
36. *Cocoon* supports *JDBC*
37. *Cocoon* supports *Velocity*

38. *Cocoon* supports *extensible Server Pages* (XSP)

39. *Cocoon* worksWith *EJB*, *JMS* which are *J2EE* implementations

40. *Cocoon* allows to build *portlets*

41. *Cocoon* supports *Lucene*

42. *Cocoon* supports object-relational frameworks such as *Hibernate*, *OJB*

43. *Cocoon* supports Java data binding frameworks *Castor*, *Betwixt*

44. *Cocoon* provides output filter *XML*, *HTML*, *XHTML*, *PDF*, *ODF*, *Excel*, *RTF*, *Postscript*, *Flash*, *SVG*, *MIDI*, *ZIP*

45. *Cocoon* provides transformations with *XSLT* and *STX*

46. *Cocoon* requires *Java 1.4.2*

47. *Cocoon* requires a Java servlet container version 2.2 like *Tomcat*, *Jetty*, *JBoss*, *JRun*, *Websphere*, or *Weblogic*

Quality Aspect

48. *Cocoon* doucmentation quality is *good*

A.4.2 Relating

49. *Cocoon* is a *PipesAndFilter* style

50. *Cocoon* data and control binding *RunTime* and *InvocationTime*

A.4.3 Codification

1 *Cocoon* instance of *MiddlewareOrientedArchitectureFamily*

1 *Cocoon* supportsApplicationType *WebApplication*

2,49 *Cocoon* hasParent *PipesAndFilter*

4,6 *CocoonPipelineMatcher* providesFeature *PipelineSelection*

5,6 *CocoonPipelineSelector* providesFeature *PipelineSelection*

7 *CocoonPipelineAction* providesFeature *RuntimeConfiguration*

9,12 *CocoonTransformer* providesFeature *SAXEventProcessing*

9,12 *Cocoon* hasComponentType *CocoonTransformer* hasParent *Transducer*

10 *Cocoon* hasConnectorType *CocoonSaxConnector* hasParent *Stream*

11 *Cocoon* hasComponentType *CocoonGenerator* hasParent *Transducer*

13 *Cocoon* hasComponentType *CocoonSerializer* hasParent *Transducer*

15,34,40 *Cocoon* supportsTechnology *JSR168Container*

16 *Cocoon* providesFeature *MultiChannelWebPublishing*

17 *Cocoon* providesFeature *StaticContentGeneration*

18 *Cocoon* hasPackage *CocoonAuthentication*

19,20,21 *CocoonAuthentication* supportsTechnology *Authentication*

23 *Cocoon* providesFeature *SessionManagement*

24,46 *Cocoon* requires *Java*

24 *Cocoon* supportsProgrammingLanguage *Java*

25,29,30,47 *Cocoon* requires *Tomcat*, *Jetty*, *JBoss*, *JRun*, *Websphere*, or *Weblogic*

26,27,31 *Cocoon* supportsTechnology *Spring*, *Avalon*

32 *Cocoon* providesTechnology *FlowControl*

33 *Cocoon* providesTechnology *Internationalization*

35 *Cocoon* providesFeature *XMLProcessing*

35 *XMLFile*, *XMLService*, *XMLDatabase* instance of *XMLTechnology*

35 *XMLTechnology* subclassof *Technology*

36 *Cocoon* supportsTechnology *JDBC*

37 *Cocoon* supportsTechnology *Velocity*

38 *Cocoon* supportsTechnology *XSP*

39 *Cocoon* supportsTechnology *JMS*, *EJB*

41 *Cocoon* supportsTechnology *Lucene*

42 *Cocoon* supportsTechnology *Hibernate*, *OJB*

42 *Hibernate* providesFeature *ObjectRelationalMapping*

43 *Cocoon* supportsTechnology *Castor*, *Betwixt*

44 *CocoonSerializer* supportsOutputFormat *XML*, *ODF*, *Excel*, *RTF*, *HTML*, *XHTML*, *Postscript*, *PDF*, *Flash*, *SVG*, *MIDI*, and *ZIP* instance of *FileFormat*

45 *Cocoon* supportsTechnology *XSLT*

45 *Cocoon* supportsTechnology *STX*

48 *Cocoon* documentationQuality *good*

50 *Cocoon* dataBindingAt *RunTime*, *InvocationTime*

50 *Cocoon* controlBindingAt *RunTime*, *InvocationTime*

The phrases 3, 8, 14 and 28 are not codified because they are outside the scope of the knowledge base. Phrase 3 describes a property of the *Cocoon* style itself and is therefore already present through the ADL formalization. Phrase 8 describes the overall concept of a pipeline and 14 the specific structure of pipelines in *Cocoon* architectures which are both described in the ADL formalization. Phrase 28 defines the term *block* as a synonym for *package*, however the term is never used in the rest of the description. Therefore it is skipped.

A.4.4 Formalization

```
Individual(PipelineSelection type(Feature))

Individual(CocoonPipelineMatcher type(ComponentType)
   value(providesFeature PipelineSelection)
)
Individual(CocoonPipelineSelector type(ComponentType)
   value(providesFeature PipelineSelection)
)
Individual(RuntimeConfiguration type(Feature))
Individual(CocoonPipelineAction type(ComponentType)
   value(providesFeature RuntimeConfiguration)
)
ObjectProperty(isRelatedTo Symmetric
   domain(Feature) range(Feature))

Individual(SAXEventProcessing type(Feature)
   value(isRelatedTo XMLProcessing)
)
Individual(CocoonTransformer type(ComponentType)
   value(hasParent Transducer)
   value(providesFeature SAXEventProcessing)
)
Individual(CocoonSaxConnector type(ConnectorType)
   value(hasParent Stream)
)
Individual(CocoonGenerator type(ComponentType)
   value(hasParent Transducer)
)
```

```
Class(FileFormat partial Standard)
Individual(XML type(FileFormat))
Individual(ODF type(FileFormat))
Individual(Excel type(FileFormat))
Individual(RTF type(FileFormat))
Individual(Postscript type(FileFormat))
Individual(HTML type(FileFormat))
Individual(XHTML type(FileFormat))
Individual(PDF type(FileFormat))
Individual(Flash type(FileFormat))
Individual(SVG type(FileFormat))
Individual(MIDI type(FileFormat))
Individual(ZIP type(FileFormat))

ObjectProperty(supportsOutputFormat
   domain(unionOf(ComponentType Technology)
   range(FileFormat))

Individual(CocoonSerializer type(ComponentType)
   value(hasParent Transducer)
   value(supportsOutputFormat XML)
   value(supportsOutputFormat ODF)
   value(supportsOutputFormat Excel)
   value(supportsOutputFormat RTF)
   value(supportsOutputFormat Postscript)
   value(supportsOutputFormat HTML)
   value(supportsOutputFormat XTHML)
   value(supportsOutputFormat PDF)
   value(supportsOutputFormat Flash)
   value(supportsOutputFormat SVG)
   value(supportsOutputFormat MIDI)
   value(supportsOutputFormat ZIP)
)
Individual(MultiChannelWebPublishing type(Feature))
Individual(StaticContentGeneration type(Feature))
Individual(SessionManagement type(Feature))

Individual(CocoonAuthenticationPackage type(Package)
   value(supportsTechnology Authentication)
)
Individual(Tomcat type(Servlet))
Individual(Jetty type(Servlet))
Individual(JBoss type(Servlet))
Individual(JRun type(Servlet))
Individual(Websphere type(Servlet))
Individual(Weblogic type(Servlet))
Individual(ServletContainer type(Servlet)
   value(isImplementedBy Tomcat)
   value(isImplementedBy Jetty)
   value(isImplementedBy JBoss)
   value(isImplementedBy JRun)
   value(isImplementedBy Websphere)
   value(isImplementedBy Weblogic)
)

Class(XMLTechnology partial Technology)
Individual(XMLFile type(XMLTechnology))
Individual(XMLService type(XMLTechnology))
Individual(XMLDatabase type(XMLTechnology))

Individual(Hibernate type(Technology)
   value(providesFeature ObjectRelationalMapping)
)

Individual(CocoonBasicStyle type(MiddlewareOrientedFamily)
   value(hasParent PipesAndFilter)
   value(supportsApplicationType WebApplication)
   value(hasComponentType CocoonPipelineMatcher)
   value(hasComponentType CocoonPipelineSelector)
   value(hasComponentType CocoonTransformer)
   value(hasComponentType CocoonGenerator)
   value(hasComponentType CocoonSerializer)
   value(hasConnectorType CocoonSaxConnector)
   value(supportsTechnology JSR168Container)
   value(providesFeature MultiChannelWebPublishing)
   value(providesFeature StaticContentGeneration)
   value(hasPackage CocoonAuthenticationPackage)
   value(providesFeature SessionManagement)
   value(requires Java)
   value(supportsProgrammingLanguage Java)
   value(supportsTechnology Tomcat)
   value(supportsTechnology Jetty)
   value(supportsTechnology JBoss)
   value(supportsTechnology JRun)
   value(supportsTechnology Wbesphere)
   value(supportsTechnology Weblogic)
   value(requires ServletContainer)
   value(supportsTechnology Avalon)
   value(supportsTechnology Spring)
   value(providesTechnology Internationalization)
   value(providesFeature XMLProcessing)
   value(supportsTechnology JDBC)
   value(supportsTechnology Velocity)
   value(supportsTechnology XSP)
   value(supportsTechnology JMS)
   value(supportsTechnology EJB)
   value(supportsTechnology Lucene)
   value(supportsTechnology Hibernate)
   value(supportsTechnology OJB)
   value(supportsTechnology Castor)
   value(supportsTechnology Betwixt)
   value(supportsTechnology XSLT)
   value(supportsTechnology STX)
   value(documentationQuality "good")
   value(dataBindingAt RunTime)
   value(dataBindingAt InvocationTime)
   value(controlBindingAt RunTime)
   value(controlBindingAt InvocationTime)
)

Individual(CocoonFlowControl type(Technology))

Individual(CocoonWithFlowscript type(MiddlewareOrientedFamily)
   value(hasParent CocoonBasicStyle)
   value(providesTechnology CocoonFlowControl)
)
```

Listing A.4: Complete definition of the Cocoon Architecture Families and all relevant new concepts, individuals, and properties required for it

B Query Development and Use

This appendix discribes the development of queries for the evaluation of the knowledge querying technique. The queries are derived from requirements of the RegIS Online system which was analyzed in the case study form Bornhold [2006].

The knowledge querying technique (see chapter 5) introduces a five step approach, namely: *question retrieval*, *codification*, *formalization*, *querying*, and *refinement*. The execution of these five steps are documented in the sections below. The evaluation of the query development and their execution is documented in section 6.2.

B.1 Question Retrieval

The function of question retrieval steps is to determine questions and queries of the requirements (MidArch task 1B) of a MidArch instance. Supplementary the questions from the quality model (MidArch task 2B) can be used as additional queries.

The case study [Bornhold 2006] does not include a list or table of target requirements. However, some requirements could be constructed by analyzing different parts of his document which are the sections: 1.2, 3.3, 3.5, 5.1.4, 5.3, 5.3.4, 5.3.5, 5.4.1.1, and 5.4.1.2. As a result the following technical requirements could be determined:

1. The current architecture is *Cocoon*-based and uses pipelines to process requests. The technology used to render the output is either XSP or CocoonForms. These technologies shall either be used in the new architecture to minimize the migration effort, or they have to be transferred to a similar technology.

2. The current systems accesses MySQL directly in XSP scripts and Hiberante to perform persistence. The dependency on XSP and Hibernate together with the migration and cost goals determine a continued use of Hibernate or a similar technology for persistence. The direct MySQL connection can be dissolved.

3. The data administration subsystem uses workflows based on the flow control of the *Cocoon* technology. The resulting system must provide a flow control mechanism. To increase use of current system modules the same flow control technology shall be used.

4. The target system must use Java.

Beside these technical requirements, there are requirements which can determined from the goals defined in the quality model.

5. Dependability require stability of the components. Also the dependency between the components and between the components and external services affect the dependability.

6. Extendability and maintainability require that new functions can be added without rewriting parts of the core system. Therefore, the system must support a way to add components at a later time.

7. Migratability as used in [Bornhold 2006] require that old system components are used preferably instead of being replaced by new components.

Based on these requirements and descriptions referring to the current system configuration a set of statements can be formulated:

1,2 The resulting system must support *XSP* and *CocoonForms* or *XSP* and *CocoonForms* based components must be transformed in components using similar technologies

2 The resulting system must support *MySQL*

2 The resulting system must support *Hibernate* or a technology which allows migration from *Hibernate*

3 The resulting system must support *workflow* like *CocoonFlowControl*

4 The resulting system uses *Java*

6 The resulting system must support later component addition

7 The resulting system should use many old components to minimize the migration effort

The requirement 5 is not listed, because it refers to a property, which can only be determined on an architecture level. The last statement cannot directly expressed as a query for architectural styles, because the therefore information of the components dependencies must be available.

B.2 Codification

The previous retrieval process resulted in six usable statements, which can be used as queries. However, some terms in these statements are not known to the system. Therefore they have to be added before the queries are executed. The first new term is *CocoonForms* which has not been defined in the analysis of the *Cocoon* styles. *CocoonForms* is an *XMLTechnology* and *isPartOf* the *Cocoon* based styles. The second term is *MySQL* which represents a database system. *MySQL* is therefore a *Technology.* And it is supported by various *ObjectRelationalMapping* technologies, such as *Hibernate* and *JPA*.

The codification of the first five statements, after the addition of the new terms is achievable as follows:

1. *MiddlewareOrientedArchitectureFamily* supportsTechnology *XSP* and *CocoonForms* or *MiddlewareOrientedArchitectureFamily* supportsTechnology *Technology* providesFeature *SeparationOfViewAndLogic* and *Technology* providesFeature *FormHandling*
2. *MiddlewareOrientedArchitectureFamily* supportsTechnology *MySQL*
3. *MiddlewareOrientedArchitectureFamily* supportsTechnology *Hibernate*
4. *MiddlewareOrientedArchitectureFamily* providesFeature *FlowControl*
5. *MiddlewareOrientedArchitectureFamily* supportsProgrammingLanguage *Java*
6. *MiddlewareOrientedArchitectureFamily* providesFeature *Modularization*

B.3 Formalization

In the formalization step, the previously defined antecedents must be expressed in SWRL. Additionally to that, a suitable consequent with SQWRL must be specified. However, the antecedent of the first rule contains an *or* conjunction. Therefore the rule has to be split in two alternative fragments. All other antecedents can be transformed in a straightforward way. This leads to the list of rule fragments shown in listing B.1.

```
/* alternative rule fragments */
Antecedent(
    MiddlewareOrientedArchitectureFamily(?family)
    supportsTechnology(?family,XSP)
    supportsTechnology(?family, CocoonForms)
)
Antecedent(
    MiddlewareOrientedArchitectureFamily(?family)
    supportsTechnology(?family,?techA)
    supportsTechnology(?family,?techB)
    providesFeature(?techA, SeparationOfViewAndLogic)
    providesFeature(?techB, FormHandling)
)
/* list of required rule fragments */
Antecedent(
    MiddlewareOrientedArchitectureFamily(?family)
    supportsTechnology(?family,MySQL)
)
Antecedent(
    MiddlewareOrientedArchitectureFamily(?family)
    supportsTechnology(?family, Hibernate)
)
Antecedent(
    MiddlewareOrientedArchitectureFamily(?family)
    providesFeature(?family, FlowControl)
)
```

```
Antecedent(
    MiddlewareOrientedArchitectureFamily(?family)
    supportsProgrammingLanguage(?family, Java)
)
Antecedent(
    MiddlewareOrientedArchitectureFamily(?family)
    providesFeature(?family, Modularization)
)
```

Listing B.1: List of all antetedent fragments

Based on the fragments in listing B.1 two alternative rules can be formulated. The first rule with all required rule fragments including the first alternative, and the second rule with all required rule fragments including the second alternative. The consequents for these two rules are simple, because the statements just require to find styles. No restrictions are included in the consequent. In listing B.2 the two resulting rules are shown.

```
/* alternative rule fragments */
Implies(
    Antecedent(
        MiddlewareOrientedArchitectureFamily(?family)
        supportsTechnology(?family,XSP)
        supportsTechnology(?family, CocoonForms)
        supportsTechnology(?family,MySQL)
        supportsTechnology(?family, Hibernate)
        providesFeature(?family, FlowControl)
        supportsProgrammingLanguage(?family, Java)
        providesFeature(?family, Modularization)
    )
    Consequent(
        sqwrl:select(?family)
    )
)

Implies(
    Antecedent(
        MiddlewareOrientedArchitectureFamily(?family)
        supportsTechnology(?family,?techA)
        supportsTechnology(?family,?techB)
        providesFeature(?techA, SeparationOfViewAndLogic)
        providesFeature(?techB, FormHandling)
        supportsTechnology(?family,MySQL)
        supportsTechnology(?family, Hibernate)
        providesFeature(?family, FlowControl)
        supportsProgrammingLanguage(?family, Java)
        providesFeature(?family, Modularization)
    )
    Consequent(
        sqwrl:select(?family)
    )
)
```

Listing B.2: The two resulting queries

B.4 Querying

The querying worked as expected and produced two results sets. The first contained only the *CocoonWithFlowscript*. The second contained the *Spring* and the *CocoonWithFlowscript*.

B.5 Refinement

The results from the *querying* step were sufficient and the two returned middleware-oriented styles are the same as the style candidates returned by task 3B of the MidArch method.

C Setup of the Knowledge Base

The knowledge base developed for my thesis, consists of two files, which are provided on the included CD. The files are named `style-meta-data.pprj` and `style-meta-data.owl`. The first file is used by Protégé-OWL to configure the different views. The second file contains the actual knowledge base.

To use the knowledge base Protégé-OWL 3.4 has to be used, therefore a copy is available on the CD. Additionally, a jar-file containing the Jess rule engine [Jess 2008] has to be installed. The jar-file is not part of the CD, because it has to be obtained from the Jess web-page `http://www.jessrules.com/jess/download.shtml`, due to licensing issues. A trial version of Jess is available, free of charge, and is sufficient to test the knowledge base. A zero-cost license for academic use is available, but one needs to sign a detailed license contract.

The two queries developed in the evaluation process can be run in Protégé-OWL in the *SWRL Rules* view. They are named `alternativeA` and `alternativeB`. To view the results of these two queries they must be executed in SQWRLQueryTab of the *SWRL Rules* view.

To execute a rule, it must be selected in rule list of the *SWRL Rules* view. Then the query tab must be opened by clicking on the small button in the upper right, labeled **SQ**.

In the opened SQWRLQueryTab the button labeled **Run** has to be pressed which produces the desired result in a separate tab, next to the SQWRLQueryTab.

Glossary

Class

A *class* in description logic is a set defined by its properties or its individuals. A class is therefore a synonym for concept.

Concept

A *concept* is a set of properties which are shared among a group of individual terms. The individuals, however, may have different values for these common properties.

Individual

An *individual* is a element of a concept, which has a distinct set of property values for properties induced by the concept.

Noun phrase

A *noun phrase* is a linguistic structure which is used in sentences. Depending on its position, it is either a subject or an object.

Object

An *object* is an element of a sentence. In English an object can be used to describe the possessor of the subject, the recipient of an action, or the object used in an action

Predicate

A *predicate* is in logic programming a function which has none, one, or more parameters and returns only true or false. Additionally a *predicate* is used in linguistics to describe a part of a sentence which connects the subject to the object.

Subject

A *subject* is an element of a sentence. The subject is the element which is performing an action. While the object is either the recipient or the used object in the action.

Bibliography

[Alford et al. 2007] ALFORD, Ron ; CUENCA-GRAU, Bernardo ; HALASCHEK-WIENER, Christian ; HEWLETT, Daniel ; GROVE, Michael ; KALYANPUR, Aditya ; KATZ, Jordan ; KOLOVSKI, Vladimir ; MURPHY, Rick ; RUCKHAUS, Edna ; SMITH, Michael: *Pellet Reasoner*. `http://pellet.owldl.com/`. Version: October 2007

[Avalon 2004] *Apache Avalon Project*. `http://avalon.apache.org/closed.html`. Version: 2004. – closed

[Avalon 2007] *Excalibur Project, Avalon Framework*. `http://excalibur.apache.org/`. Version: 2007

[Baader et al. 2007] BAADER, Franz (Hrsg.) ; CALVANESE, Diego (Hrsg.) ; MCGUINNESS, Deborah (Hrsg.) ; NARDI, Daniele (Hrsg.) ; PATEL-SCHNEIDER, Peter (Hrsg.): *The Description Logic Handbook*. Cambridge Univ. Press, 2007

[Basili et al. 1994] BASILI, Victor R. ; CALDIERA, Gianluigi ; ROMBACH, Dieter: The Goal Question Metric Approach. In: *Encyclopedia of Software Engineering* 2 (1994), S. 528—532

[Bechhofer et al.] BECHHOFER, Sean ; HARMELEN, Frank van ; HENDLER, Jim ; HORROCKS, Ian ; MCGUINNESS, Deborah L. ; PATEL-SCHNEIDER, Peter F. ; STEIN, Lynn A.: *OWL Web Ontology Language – Reference*. `http://www.w3.org/TR/owl-ref/`

[Beckett 1999] BECKETT, Dave: *Resource Description Framework*. `http://www.w3.org/RDF/`. Version: 1999

[Berners-Lee et al. 2001] BERNERS-LEE, Tim ; HENDLER, James ; LASSILA, Ora: The Semantic Web. In: *Scientific American Magazine* (2001), May. `http://www.sciam.com/article.cfm?id=00048144-10D2-1C70-84A9809EC588EF21&page=1`

[Boley and Kifer 2007] BOLEY, Harold ; KIFER, Michael: *RIF Basic Logic Dialect*. `http://www.w3.org/TR/2007/WD-rif-bld-20071030/`. Version: October 2007. – W3C Working Draft

[Bornhold 2006] BORNHOLD, Johannes: *Migration eines regionalen Wirtschaftsinformationssystems*, Carl von Ossietzky Universität Oldenburg, Diplomarbeit, 2006

[Brickley and Guha 2004] BRICKLEY, Dan ; GUHA, R.V.: *RDF Vocabulary Description Language 1.0: RDF Schema*. `http://www.w3.org/TR/rdf-schema/`. Version: February 2004

[CEL 2008] *A polynomial-time Classifier for the description logic EL+.* `http://lat.inf.tu-dresden.de/systems/cel/`. Version: January 2008

[Cocoon 2008] *Apache Cocoon 2.2.* `http://cocoon.apache.org/`. Version: 2008

[Coenen et al. 2007] Coenen, Frans ; Leng, Paul ; Sanderson, Robert ; Wang, Yanbo J.: Statisitical Identification of Key Phrases for Text Classification. In: Perner, Petra (Hrsg.): *Machine Learning and Data Mining in Pattern Recognition*, Springer, July 2007 (5th International Conference, MLDM 2007), S. 838–853

[Dale 2000] Dale, Robert (Hrsg.): *Handbook of natural language processing.* New York : Dekker, 2000

[DIG 2006] *The new DIG interface standard (DIG 2.0).* `http://dl.kr.org/dig/interface.html`. Version: March 2006

[Dijkstra 1982] *Chapter* On the role of scientific thought. In: Dijkstra, Edsger W.: *Selected writings on Computing: A Personal Perspective.* New York, USA : Springer-Verlag New York, 1982, S. 60—-66

[Eclipse 2008] *Eclipse - An Open Development Platform.* `http://www.eclipse.org/`. Version: 2008

[Excalibur 2007] *Excalibur Project.* `http://excalibur.apache.org/`. Version: 2007

[FOLDOC 2007] *Free On-Line Dictionary Of Computing.* `http://foldoc.org/index.cgi?query=data&action=Search`. Version: September, 10th 2007

[Fowler 2004] Fowler, Martin: *Inversion of Control Containers and the Dependency Injection pattern.* `http://www.martinfowler.com/articles/injection.html`. Version: January 2004

[Friedman-Hill 2008] Friedman-Hill, Ernest J.: *Jess Language Basics.* `http://herzberg.ca.sandia.gov/jess/docs/70/basics.html`. Version: March 2008

[FSF 1991] *GNU General Public License, version 2.* `http://www.gnu.org/licenses/gpl-2.0.html`. Version: June 1991

[Gerbner et al. 1969] Gerbner, George (Hrsg.) ; R.Holsti, Ole (Hrsg.) ; Krippendorf, Klaus (Hrsg.) ; J.Paisley, William (Hrsg.) ; J.Stone, Philip (Hrsg.): *The Analysis of Communication Content:Developments in Scientific Theories and Computer Techniques.* New York : John Wiley & Sons, 1969

[Giesecke 2008] Giesecke, Simon: *Architectural Styles for Early Goal-driven Middleware Platform Selection*, Carl von Ossietzky University Oldenburg, Diss., 2008

[Giesecke et al. 2007] Giesecke, Simon ; Bornhold, Johannes ; Hasselbring, Wilhelm: Middleware-induced Architectural Style Modelling for Architecture Exploration. In: *Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), January 2007, Mumbai, India*, IEEE Computer Society Press, 2007

[Givón 1993] Givón, Talmy: *English Grammar: A Function-based Introduction.* John Benjamins Publishing Company, 1993

[GmbH 2008] GmbH, REGIO: *RegIS Online website.* `http://www.regis-online.de`. Version: 2008

[Gruber 1993] Gruber, Thomas R.: A Translation Approach to Portable Ontology Specifications. In: *Knowledge Systems Laboratory – Technical Report KSL 92-71* (1993), April. – revised edition

[Haegeman and Guéron 1999] Haegeman, Liliane M. V. ; Guéron, Jacqueline: *English Grammar: a Generative Perspective.* Blackwell, 1999

[Hirtle et al. 2006] Hirtle, David ; Boley, Harold ; Grosof, Benjamin ; Kifer, Michael ; Sintek, Michael ; Tabet, Said ; Wagner, Gerd: *Schema Specification of RuleML 0.91.* `http://www.ruleml.org/0.91/`. Version: August 2006

[Horn 1951] Horn, Alfred: On sentences which are true of direct unions of algebras. In: *Journal of Symbolic Logic* 16 (1951), March, Nr. 1, S. 14–21

[Horridge et al. 2004] Horridge, Matthew ; Knublauch, Holger ; Rector, Alan ; Stevens, Robert ; Wroe, Chris: *A Practical Guide To Building OWL Ontologies Using The Protege – OWL Plugin and CO-ODE Tools.* Edition 1.0. The University Of Manchester, 2004

[Horrocks et al. 2004] Horrocks, Ian ; Patel-Schneider, Peter F. ; Boley, Harold ; Tabet, Said ; Grosof, Benjamin ; Dean, Mike: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* `http://www.w3.org/Submission/SWRL/`. Version: May 2004

[Jackson and Moulinier 2002] Jackson, Peter ; Moulinier, Isabelle: *Natural language processing for online applications : text retrieval, extraction and categorization.* Benjamins Publishing Company, 2002

[Jena 2008] *Jena –- A Semantic Web Framework for Java.* `http://jena.sourceforge.net/`. Version: March 2008

[Jess 2008] *Jess, the Rule Engine for the Java Platform.* `http://herzberg.ca.sandia.gov/`. Version: Febuary 2008

[Johnson et al. 2008] Johnson, Rod ; Hoeller, Juergen ; Arendsen, Alef ; Sampaleanu, Colin ; Harrop, Rob ; Risberg, Thomas ; Davison, Darren ; Kopylenko, Dmitriy ; Pollack, Mark ; Templier, Thierry ; Vervaet, Erwin ; Tung, Portia ; Hale, Ben ; Colyer, Adrian ; Lewis, John ; Leau, Costin ; Fisher, Mark ; Brannen, Sam ; Laddad, Ramnivas: *The Spring Framework - Reference Documentation.* 2008 `http://www.springframework.org/documentation`

[Johnson 2002] Johnson, Roh: *Expert One-on-One J2EE Design and Development.* Wrox, 2002

[Krippendorff 1980] KRIPPENDORFF, Klaus: *Content Analysis: An Introduction to Its Methodology.* Sage Publications, 1980

[Krämer 1999] KRÄMER, Sybille: Information. In: SANDKÜHLER, Hans J. (Hrsg.) ; PÄTZOLD, Detlev (Hrsg.): *Enzyklopädie Philosophie.* Meiner, 1999 (Band 1 A–N), S. 636 – 640

[Lehmann 2007] LEHMANN, Jens: Hybrid Learning of Ontology Classes. In: PERNER, Petra (Hrsg.): *Machine Learning and Data Mining in Pattern Recognition*, Springer, July 2007 (5th International Conference, MLDM 2007), S. 883–898

[Mayring 2000] MAYRING, Philipp: Qualitative Content Analysis. In: *Forum Qualitative Social Research* Volume 1 (2000), June, Nr. No 2. `http://www.qualitative-research.net/fqs-texte/2-00/2-00mayring-e.pdf`

[McBride 2004] MCBRIDE, Brian: *RDF/XML Syntax Specification.* `http://www.w3.org/TR/rdf-syntax-grammar/`. Version: February 2004

[McGuinness and van Harmelen 2004] MCGUINNESS, Deborah L. ; HARMELEN, Frank van: *OWL Web Ontology Language Semantics and Abstract Syntax.* `http://www.w3.org/TR/owl-semantics/`. Version: 2004

[Metha 2004] METHA, Nikunj R.: *Composing Style-base Software Architectures from Architectural Primitives*, Faculty of the Graduate School, University of Southern California, Dissertation, 2004

[Mittelstraß 1980a] Ontologie. In: MITTELSTRASS, Jürgen (Hrsg.) ; BLASCHE, Siegfried (Hrsg.) ; CARRIER, Martin (Hrsg.) ; WOLTERS, Gereon (Hrsg.): *Enzyklopädie Philosophie und Wissenschaftstheorie.* second edition. Metzler, 1980 (Band 2 H–O), S. 1077–1079

[Mittelstraß 1980b] Wissen. In: MITTELSTRASS, Jürgen (Hrsg.) ; BLASCHE, Siegfried (Hrsg.) ; CARRIER, Martin (Hrsg.) ; WOLTERS, Gereon (Hrsg.): *Enzyklopädie Philosophie und Wissenschaftstheorie.* second edition. Metzler, 1980 (Band 4 Sp–Z), S. 717–719

[Monroe et al. 1997] MONROE, Robert T. ; KOMPANEK, Andrew ; MELTON, Ralph ; GARLAN, David: Architectural Styles, Design Patterns, and Objects. In: *IEEE Software* (1997), S. 43–52

[Osman 2004] OSMAN, Hesham: *A knowledge-enabled system for coordinating the design of co-located urban infrastructure.* `http://individual.utoronto.ca/hesham/Ontology/IPDFull.owl`. Version: 2004

[Osman and El-Diraby 2006] OSMAN, Heshman M. ; EL-DIRABY, Tamer E.: Ontological Modeling of Infrastructure Products and Related Concepts. In: *Transportation Research Record* (2006), S. 159–167

[Pahl et al. 2007] Pahl, Claus ; Giesecke, Simon ; Hasselbring, Wilhelm: An Ontology-based Approach for Modelling Architectural Styles. (2007)

[Polanyi 1966] Polanyi, Michael: *The Tacit Dimension.* Doubleday & Co, 1966

[Possner 1998] Possner, Roland: Pragmatics. In: Bouissac, Paul (Hrsg.): *Encyclopedia of Semiotics.* Oxford University Press, 1998, S. 515

[Protégé 2008] *Protege-OWL Wiki, What are the current limitations of the SWRL Rule Engine Bridge?* `http://protege.cim3.net/cgi-bin/wiki.pl?SWRLRuleEngineBridgeFAQ#nid6QL`. Version: January 2008

[Prud'hommeaux and Seaborne 2008] Prud'hommeaux, Eric ; Seaborne, Andy: *SPARQL Query Language for RDF.* `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`. Version: January 15 2008

[Racer 2008] *RacerPro Reasoner.* `http://www.racer-systems.com/`. Version: 2008

[RedHat 2008] RedHat: *Hibernate.* `http://www.hibernate.org/`. Version: 2008

[RIF-WG 2008] *Rule Interchange Format (RIF) Working Group.* `http://www.w3.org/2005/rules/wiki/RIF_Working_Group`. Version: April 2008

[Rosengren 1981] Rosengren, Karl E. (Hrsg.): *Advances in Content Analysis.* Bd. Volume 9. Beverly Hills : Sage Publications, 1981

[Schnelle 1976] Schnelle, H.: Information. In: Ritter, Joachim (Hrsg.): *Historisches Wörterbuch der Philosophie.* Schwabe Verlag, 1976 (Band 4 I–K), S. 356–357

[Shaw and Clements 1997] Shaw, Mary ; Clements, Paul: A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In: *Proceeding of the 21st International Computer Software and Applications Conference*, 1997, S. 6 – 13

[Spivey 1998] Spivey, J. M.: *The Z Notation: A Reference Manual.* Oxford: University of Oxford, 1998

[Spring 2008] *Spring Framework.* `http://www.springframework.org/`. Version: March 2008

[SQWRL 2007] *Semantic Query-Enhanced Web Rule Language.* `http://protege.cim3.net/cgi-bin/wiki.pl?SQWRL`. Version: November 19th 2007

[Strobl 1996] Strobl, Reiner (Hrsg.): *Wahre Geschichten? : zu Theorie und Praxis qualitativer Interviews ; Beiträge zum Workshop Paraphrasieren, Kodieren, Interpretieren ... im Kriminologischen Forschungsinstitut Niedersachsen am 29. und 30. Juni 1995 in Hannover.* Nomos-Verlagsgesellschaft, 1996

[Sun Microsystems 2008a] Sun Microsystems: *Java Persistence API.* `http://java.sun.com/javaee/technologies/persistence.jsp`. Version: 2008

[Sun Microsystems 2008b] SUN MICROSYSTEMS: *Java Sever Faces Technology.* `http://java.sun.com/javaee/javaserverfaces/`. Version: 2008

[Sun Microsystems 2008c] SUN MICROSYSTEMS: *The Source for Java Developers.* `http://java.sun.com`. Version: 2008

[Swan 2005] SWAN, Michael: *Practical English Usage.* third edition. Oxford, UK : Oxford University Press, 2005

[Tsarkov and Horrocks 2007] TSARKOV, Dmitry ; HORROCKS, Ian: *FaCT++ Reasoner.* `http://owl.man.ac.uk/factplusplus/`. Version: May 2007

[W3C 2007] *W3C Semantic Web Activity.* `http://www.w3.org/2001/sw/`. Version: 2007

[WOP] *The Web of Patterns Project.* `http://www-ist.massey.ac.nz/wop/`

Declaration

This thesis is my own work and contains no material that has been accepted for the award of any other degree or diploma in any university.

To the best of my knowledge and belief, this thesis contains no material previously published by any other person except where due acknowledgment has been made.

Place, Date, Signature