INGENJÖRSHÖGSKOLAN
HÖGSKOLAN I JÖNKÖPING

Application patterns for ontology based applications

Thomas Albertsen

Final Thesis 2006
Information Technology

Application patterns for ontology based applications

Thomas Albertsen

Tutor: Eva Blomqvist

Size: 20 points

Date: 2006-09-18

Arkiveringsnummer:

Abstract

Software patterns have been proven as a valuable way to storing a repeatable solution to a commonly occurring problem in software design. A pattern is not a finished design that can be directly formed into program code; instead it is a description how to solve a problem that may occur in many situations.

In the ontology community very little research have been made in producing high-level patterns where the solution shows how an architecture of an ontology based software might look like.

In this thesis the results of examining how high-level patterns of this type relates to other types of patterns are given and how these patterns would be described are formulated.

Keywords

Ontology, patterns, software architechture, architecture patterns

Table of contents

1 Introduction

The use of ontologies to bridge the semantic gap between the syntactical representation of information and the conceptualization of that information has been proven as an effective way to facilitate communication and to structure information. In the last decade this field has matured and a lot of different applications that uses this technology have been developed and more is likely to follow. The Semantic Web, where knowledge is stored through the use of machine-processable metadata, is perhaps the most prominent and most known uses of this technology, and makes it possible to create applications that can make the enormous information source that the Internet constitutes more manageable. But the use of ontologies are not limited to use on the Web. With the vast amount of information present in most organizations or companies there is a need for knowledge management systems and corporate memories where information can be structured, integrated and analyzed. This is another example of ontology based applications.

Patterns have also developed into an important field of computer science and have proven to be a good way of capturing knowledge of how to solve recurring design problems in a specific context. The pattern also gives an insight into the rationale behind the solution and shows the consequences of implementing the pattern by reflecting the advantages and disadvantages of the solution. The solutions shown in patterns should be mature proven solutions that are implemented several times.

With the large amount of applications developed that are based on ontologies, there is a need to capture the essence of the solutions used to create good applications so that other developers can use this information to help them find and understand good solutions in this field. This can facilitate the creation of new software that is based on ontologies. There has been very little research done in developing high-level patterns of ontology based applications. Ontology Application Patterns are high-level patterns that aim to capture the architecture of an application or system that are based on ontologies.

1.1 Purpose

The purpose of this thesis is to find a definition of Ontology Application Patterns and examine how these patterns relate to other types of software patterns. These patterns could help the development of software by letting developers understand solutions to general problems in this field. Since these are high-level patterns, the solutions are usually architectures of applications and by understanding them, designing architectures can be facilitated.

Another purpose of this thesis is to examine what the benefits of using patterns of this type is, i.e. what the advantages and disadvantages of storing knowledge in a pattern form and what the consequences of using this information might be.

1.2 Problem formulation

This thesis has two general purposes and this gives us two problem formulations where the first addresses the question of if it is possible to create patterns of this type and how they might look like. This first problem formulation is:

Examine if it is possible to find a definition of high-level patterns of ontology based applications and examine how these patterns relate to other types of software patterns.

The other problem formulation is concerning the question of in what areas these patterns might be of importance and also what risks and advantages that might be involved in using patterns of this type when designing software. This problem formulation is:

Examine in what areas these patterns can be applicable and also examine the strengths and weaknesses of using patterns of this type.

1.3 Scope

The scope of this thesis is to investigate what Ontology Application Patterns really are and how they relate to other types of software patterns. It is not to create such patterns or to validate existing patterns.

1.4 Disposition

The rest of the thesis is structured as follows, in chapter two the theoretical background of the subject is presented to make the rest of the thesis easier to understand. Chapter three describes the method used during this work. In chapter four the process of finding the definition of the patterns and the benefits of them are described. Chapter five describes the results.

2 Theoretical background

In this chapter the relevant theoretical background for this thesis is explained.

2.1 Ontologies

The word ontology has more definitions than one definition which is a little ironic since ontologies are used to make clear and distinct statements. The word ontology was first used in philosophy and means "The metaphysical study of the nature of being and existence". In more plain words: The study of how do thoughts, words and things really relate to each other. The meaning that the word has in computer science is derived from the philosophical meaning. [1]

Probably the most common definition of ontology in the field of computer science is "a formal, explicit specification of a shared conceptualization"[2]. Conceptualization is an abstract model of how people think and when this model is given an explicit specification, we give names and meanings to the concepts and relations present in this abstract model. When the specification is formal, the meaning is that a language is used that have well understood formal properties so no ambiguities that natural language tend to give exist. This usually means some kind of logic based language is used.

Why are ontologies important in the first place? What the ontology provides is a specification of the concepts in the domain without the ambiguities that natural language gives. Since the meaning of the concepts and relations in an ontology are specified, these ambiguities are removed. This gives the users of the ontology, humans and computers, a shared vocabulary both syntactically and semantically [3].

In an ontology concepts and relations between concepts are defined. Rules for how these concepts may be related are also defined. Concepts are anything that it is possible to say something about. It can be something physical and real but it can also be something abstract and fictitious. When a concept is defined, what is really done is to create a meaning that this particular concept stands for in the domain that is being described. To make it possible to refer to a concept, it is necessary to put a label on it, i.e. to connect a term to the concept. A domain is the area that are of interest to the parties involved in developing the ontology. This can be information concerning an organization or what wines that is produced in France [1].

The concepts usually have attributes that describes them. For example the concept "person" usually has an attribute "name" and perhaps "phone number" and "address". Attributes are sometimes called slots or properties [1].

Between concepts in the ontology exists relations. Perhaps the most usual relation is “is a” which is used to describe a hierarchy of concepts. In a hierarchy there are subclasses which inherit the attributes that the parent concept has. This is generally called taxonomy and it is very common that ontologies are structured this way although it is not a necessity [3].

As mentioned before, only concepts that are relevant to the domain of interest are defined in the ontology. Such ontologies are called domain ontologies. Some concepts are more general and might be present in all ontologies. One idea how to avoid that these concept have to be defined over and over is to introduce top-level ontologies or upper level ontologies. In these ontologies general concepts that can be reused are defined. When new ontologies are constructed the general concepts from these ontologies are included. At this moment some ontologies that could function as top-level ontologies exist but none of them are used as some sort of standard. An examples of a suggested top-level ontology is the Suggested Upper Merged Ontology developed by Standard Upper Ontology Working Group [4]. There are also ideas concerning domain dependent top-level ontologies that have concepts that are general in a specific domain. This would also limit the work when new ontologies are constructed since a top-level ontology constitutes the foundation for the ontology being built [3].

If the operational data is included in the ontology, i.e. there are individual instances of concepts in the ontology, a knowledge base has been created. So in the ontology about wines the concepts of wine with the subclasses red and white wine are defined. If an extra level where there are instances of a lot of individual red and white wines exist, this ontology constitutes a knowledge base [3].

When an ontology is constructed within an organization it also makes the information concerning the domain more unambiguous since the task of structuring the information demands strict definition of the concepts. This means that the information can be reused and assumptions that have been made about the domain will be explicit, i.e. the process of creating an ontology not only structures the information but also clears up misunderstandings and identifies gaps in the information [5].

Although ontologies have been proven to be very useful there are some problems and issues that have to be taken into concern when using them. The process of classifying concepts is inherently problematic since not everyone has the same idea of what a certain concept refers to. When an ontology is developed this is one of the big problems to overcome. The conceptualization must be shared among its users and this is problematic if the users are a very heterogeneous group. This “common idea” of what a concept stands for is called the ontological commitment. There is also the issue of maintaining the ontology. The stored information must continually be changed to mirror the real world or the ontology will be useless [3].

2.1.1 Ontology languages

To be able to formulate ontologies some sort of formal syntax is needed and there are several languages which can be used. There are a large number of different languages that can be used when constructing ontologies. Some of them have been around for a long time like CycL [6] , Ontolingua [7] and F-logic [8]. Other languages are developed more recently and are developed to be used on the Internet in order to accomplish the vision of the Semantic Web [9]. Examples of these languages are RDF(S) (Resource Description Framework and Resource Description Framework Schema) [10][11], OIL (Ontology Inference Layer) [12] and OWL (Web Ontology Language) [13].

2.1.2 Ontology integration

Since there is more than one ontology present, there are numerous scenarios when there are reasons to integrate two or more ontologies with each other. This can be done in several different ways; merging, mapping and aligning are probably the most common used approaches. This terminology is defined in [14] in the following way:

- Merging (also sometimes called integrating) creates a new ontology from two or more ontologies that have overlapping parts.

- Aligning brings two or more ontologies into a mutual agreement, which makes them consistent and coherent with each other.

- Mapping relates similar concept concepts and relations from two or more ontologies to each other by specifying the correspondence to each other.

The differences in the meaning of a concept described by two or more ontologies are called mismatches. These mismatches can be divided into two levels, language level and ontology level. When mismatches occur in the language level, the differences between the ontologies are in how the mechanism for defining classes and relations works. A good example of this is the syntactical difference between ontologies defined in RDF Schema and ontologies defined in LOOM. Another example of language level mismatch is difference in the expressivity of the language, where one language can express things that can not be formulated when using another language. An example of this is that some languages can express negation and others can not [14].

Ontology level mismatches occur when ontologies describe overlapping domains. These mismatches are not related to the ontology language, instead they are differences in how the domain is modeled. Examples of this is terminological mismatches such as synonyms, where different terms are used to describe the same concept, and homonyms where the same term is used to describe different concepts [14].

Other mismatches on this level are scope and model coverage and granularity, which are called conceptualization mismatches. The scope mismatch is when two classes represent the same concepts but do not have the same instances and model coverage and granularity is what part of the domain that are modeled or with what granularity the domain is modeled [14].

Other differences can be different paradigms to represent basic concepts as time or space. Also there can be differences in concepts descriptions, i.e. how the concepts are modeled in the ontology. Examples of this can be where distinctions between features in an is-hierarchy is made, in some ontologies these are put high up in the hierarchy, in others on a low level. These types of differences are called explication mismatches. Ontologies can also differ in encoding, where values are stored in different formats [14].

To be able to integrate the ontologies these mismatches has to be solved, both on the language level and the ontology level. On the ontology level different approaches have been proposed and are implemented in some tools. One suggestion is algorithms based on heuristic matching. These can be divided into two types, linguistic based matches and structural and model similarities [14]. Linguistic based matching means that terms that have the same word stem or are closely related in word-stem are matched. The other type, structural and model similarities is when the structures of the ontologies are analyzed and matched. [14]

Another approach is to have a the same top level ontology. The more specialized ontologies are mapped to this ontology and this removes the mismatches. These top level ontologies can be more general, such as the IEEE Standard upper Ontology [15], or more specific such as the shared ontologies in KRAFT [16]. The problem with this approach is that it requires manual mapping to the shared ontology [14].

A totally different approach to mapping is language game, where a common ontology emerges through interactions between the communicating peers. The general idea behind the language game is rather simple. The peers communicate pair wise and in each of these sessions there are a speaker and a hearer. The speaker picks an object and put label on it. The hearer decodes the label and picks an object that seems the most reasonable. This object is sent to the hearer and if the object is the same as the one the object intended, the game has succeeded and the label used gets a closer connection to the object. If the hearer does not understand the label, the speaker gives the hearer the object and a connection between the object and the label is made. The idea is that after a lot of these games, all peers will tend to have the same labels for the same objects. This type of language game have been used to make robots that uses cameras build a common language [17].

2.1.3 Ontology dimensions

To be able to describe a whole ontology there is a need for some sort of characteristics of the ontology. These characteristics can also be called the dimensions of the ontology. A try to formulate such characteristics has been done in [18] and the dimensions described there are:

Level of Authoritativeness: This is a measure of how authoritative the ontology is of the area it describes. If the author of the ontology is the organization that is responsible for specifying the conceptualization, then the ontology might define the knowledge in the area, then this is clearly a highly authoritative ontology. [18]

Source of Structure: If the ontology is developed externally from the application that will use it and changes are made systematically are called transcendent, while ontologies where the structure comes directly from the application that will use the ontology are called immanent. This means that the structure might change radically depending on what happens in the use of the application. [18]

Degree of Formality: Degree of formality refers to the level of formality of the specification of the conceptualization, this could be from highly informal or taxonomic ontologies, to semantic networks, that may include complex subclass/superclass relations but nonformal axiom expressions, to highly formal ontologies that include axioms that explicitly define concepts. [18]

Model Dynamics: This concern the rate of how changes in the ontology are made. From the extreme where the ontology is stable and rarely or never change, to very volatile ontologies that changes very often. [18]

Instance Dynamics: This dimension is closely related to Model Dynamics but concerns the instances of the ontology. [18]

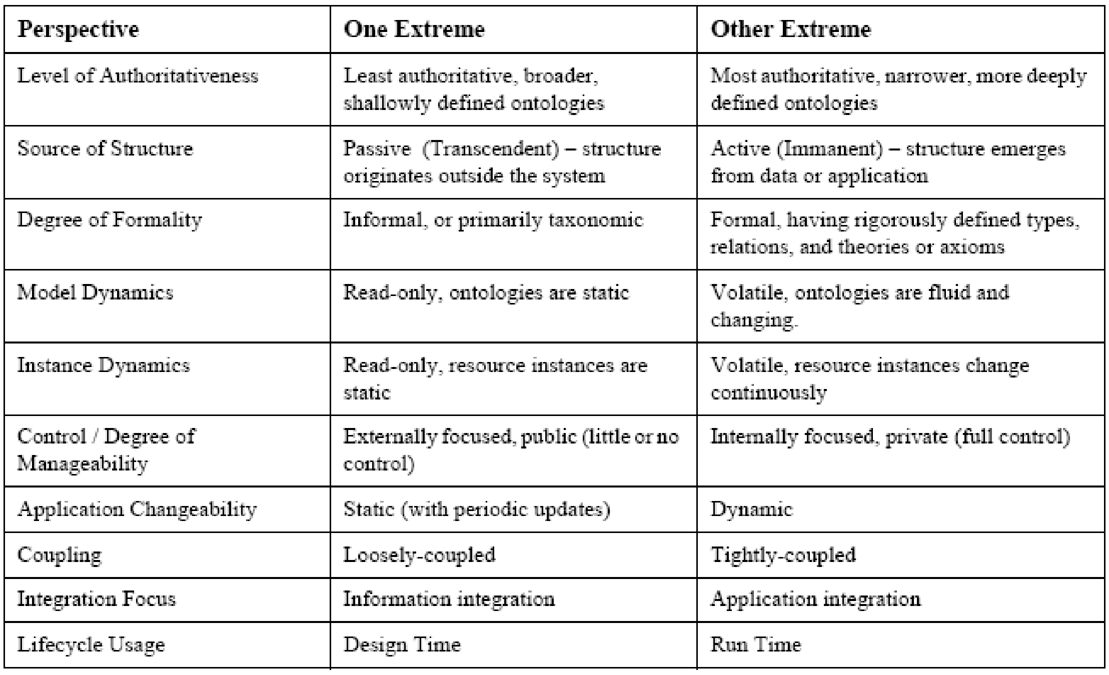
Control / Degree of Manageability: This dimension considers who decides when and how much change to make to an ontology. One extreme is that the author of the ontology has the sole decision on changes, and the other extreme of course is that the ontology must change based on outside parties. This is called internal and external focus. [18]

Application Changeability: The applications that use the ontology might be on one extreme developed only once and on the other dynamically during run time. [18]

Coupling: This dimension describes how closely coupled applications committed to shared ontologies are to each other. The applications in an e-commerce exchange are *tightly coupled*, since they must interoperate at run time. At the other extreme, applications using the Periodic Table may have nothing in common at run time. They are *loosely coupled*, solely because they share a component. [18]

Integration Focus: This dimension describes the focus of the ontology concerning integration. One extreme is the ontology that specify the structure of interoperation but not the content. This is called application integration. The other extreme is the information integration where the structure of the information is described. [18]

Lifecycle Usage: In some cases the ontology is only used in the specification or design of an application but is never used during run time. The other extreme is for example that every message sent in an application is verified so it conforms to an ontology. In [18] an overview of the extremes of the dimensions are given



Perspective	**One Extreme**	**Other Extreme**
Level of Authoritativeness	Least authoritative, broader, shallowly defined ontologies	Most authoritative, narrower, more deeply defined ontologies
Source of Structure	Passive (Transcendent) – structure originates outside the system	Active (Immanent) – structure emerges from data or application
Degree of Formality	Informal, or primarily taxonomic	Formal, having rigorously defined types, relations, and theories or axioms
Model Dynamics	Read-only, ontologies are static	Volatile, ontologies are fluid and changing.
Instance Dynamics	Read-only, resource instances are static	Volatile, resource instances change continuously
Control / Degree of Manageability	Externally focused, public (little or no control)	Internally focused, private (full control)
Application Changeability	Static (with periodic updates)	Dynamic
Coupling	Loosely-coupled	Tightly-coupled
Integration Focus	Information integration	Application integration
Lifecycle Usage	Design Time	Run Time

Table 1, *Extremes of the ontology dimensions.* [18]

2.1.4 Ontology services.

Ontologies are introduced in the system to provide some sort of functionality. This functionality can be seen as a service that the ontology provides to the rest of the system. In [19] examples of such services are given:

- Query access to ontologies, this service enables the user of the ontology to specify a query string and get a result from the ontology where the variables in the query are associated with the possible classes or relationships [19].
- Generation of ontology views, this service provides the possibility to get a subset of an ontology based on a specification provided by the user [19].

- Translation of ontologies from one language to another, this service gives the possibility to translate an ontology from one representation language to another [19].

- Management of multiple ontologies, this service provides the possibility to perform mapping between ontologies, merging of ontologies and versioning of ontologies [19].

- Reasoning on ontologies, this service gives the possibility to do semantically-based manipulations of one ormore ontologies. This service can be used for checking the ontology's consistency or computing the subsumption hierarchy of an ontology [19].

2.2 Ontology applications

There are a lot of applications that uses ontologies in some way. In [20] a classification of four ontology application scenarios is made:

Neutral authoring: In this scenario the ontology is authored in a single language and is converted into an appropriate form for each system that uses is. The benefits of this approach are knowledge reuse, improved maintainability and long term knowledge retention [20].

Ontology as Specification: An ontology of a given domain is created and used as a basis for specification and development of some software. Benefits of this approach include documentation, maintenance, reliability and knowledge (re)use [20].

Common Access to Information: Information is required by one or more persons or computer applications, but is expressed using unfamiliar vocabulary, or in an inaccessible format. The ontology helps render the information intelligible by providing a shared understanding of the terms, or by mapping between sets of terms. Benefits of this approach include inter-operability, and more effective use and reuse of knowledge resources [20].

Ontology-Based Search: An ontology is used for searching an information repository for desired resources (e.g. documents, web pages, names of experts). The chief benefit of this approach is faster access to important information resources, which leads to more effective use and reuse of knowledge resources. [20]

In [21] four scenarios for ontology based applications are described. These are made from a business perspective, i.e. the scenarios are based on a business area, not on what technology that is used within the applications. These scenarios are:

Corporate Intranet and Knowledge Management: This scenario concerns knowledge management. In the past, knowledge management has focused on management of knowledge stored within text documents. In the future, the possibility to use ontologies to specify a shared conceptualization of an application domain and in this way provide a foundation to define metadata that have a precisely defined semantics and are machine-processable give the possibility to create solutions that are based on semantically grouped information [21].

E-Commerce: Electronic Commerce is based on the exchange of information between stakeholders using some sort of communication infrastructure. [21] define two scenarios within this scenario, Business-to-Customer (B2C) and Business to Business (B2B). The B2C applications enable customers to find offers that meets their demands and service providers the possibility to reach their customers. The B2B applications make it possible for companies to exchange information concerning services already agreed upon or perhaps investigating new business proposals between the companies. In the past the information exchange has been more or less strictly of the first type but adding ontologies to he systems may give the possibility to make the information exchange more powerful and less static which could give for example the possibility to have the B2B-applications find appropriate partners. [21]

Information retrieval: These types of applications are designed to find relevant information that meets the information demand of a user. The ontologies are here used to guide the search so that the application returns more relevant results. [21]

Portals and Web communities: Portals are websites that provide information on a semantic basis. These portals usually have a backbone consisting of a knowledge warehouse, i.e. the ontology and the knowledge base and an inference mechanism. The system also has a front end that the users can interact with. These users may be other applications such as software agents. The human users can both be general users that only can access the data or community users which can contribute data. [21]

Examples of applications from the group Common access to data are KRAFT[16], InfoSleuth[22], DOME[23], Toronto Virtual Enterprise (TOVE) [24] and MOMIS[25]. KRAFT is an agent based system that enables integration of heterogeneous data sources. In the system there are three types of agents; wrappers, mediators and facilitators. Wrappers are proxies for the data sources and user agents. The wrapper is essentially a translator between the communication language used in the KRAFT system and the language used in the data source. The mediators are internal knowledge processing agents in the system. Typical tasks for these agents are filtering, sorting, and fusing knowledge obtained from other agents. The mediators are the matchmakers that enable the communication between the other agents in the system. [16]

Each data source has a local ontology that describes the information stored in the source. The concepts and relations in this ontology are mapped to concepts and relations in a shared ontology .When a message is sent, the content of the message are terms that are defined in the shared ontology[16].

The InfoSleuth system is pretty similar to the KRAFT system. Each resource has a Resource agent that connects the source to the system. The resource agent maps the information in the source to an ontology. There are usually several ontologies defined in the system. The resource agent then can translate queries that are written in the format described by the ontology to the internal format used in the data source. The users interact with the system using a user agent that translates the queries made by the user to a format described by the appropriate ontology. The system also has other agents that are necessary to find the appropriate resources and make advanced queries possible[22].

In the DOME approach three types of ontologies are used; resource ontologies, shared ontologies and application ontologies. The resource ontologies specifies the structure of the content in a data source. The shared ontologies are general ontologies over the domain and these are specialized to application ontologies which describe the domain for a certain application or group of users. The resource ontologies are mapped to the application ontologies and this makes interoperability within the system possible [23].

The MOMIS approach does not rely on mapping a local ontology to a shared one. Instead the general idea is to integrate all data sources into a single ontology that are called the common thesaurus which can be used to search and collect the information. The shared ontology is built from the schemas over the information that are present. MOMIS is intended to integrate structured or semi-structured information, not unstructured like web pages. The integration is semi automatic. Each data source has a wrapper that connects the source to the system and acts like a translator. A mediator connects the wrappers to the shared ontology and the users interact with the ontology to collect information. [25]

The approach of having a single shared ontology to integrate the data sources is also used in TOVE, where one ontology has been developed to provide a shared vocabulary for applications to use to understand each other. In this system there are also agents that acts as a translator between the applications and the ontology. [24]

Almost all ontology based application uses some sort of query based information retrieval. Using ontologies in information retrieval systems gives the obvious benefit of higher precision, using context based searching. However, it is also possible to deal with other common issues in information retrieval systems; information quality and user adaptation [21].

Information quality determines the quality by answering a few questions. Is the information up to date? Are there any different versions of the information? Do it exist any conflicting information?

Most users or groups of users use different vocabularies. Also, they usually want a different level of specialization of the information retrieved. This can be called user adaptation. With user ontologies this can be set with different templates, and the ontology can also adapt with user interaction. The information retrieval component in InfoSleuth [22] uses user agents to adapt to different user's vocabulary [26].

There are a number of applications designed to search information stored in one or several repositories, this could be an intranet or WWW. In [27] three different scenarios for searching are defined:

- Central index using one ontology
- Individual ontologies for each data source
- Linguistic ontologies

Applications using ontologies as a central index usually have a group of users defining the ontology or a number of ontologies for the domain. This method was used in TOVE. A set of ontologies defines the domains of enterprises, each application that uses TOVE has to adapt to use the terminology of TOVE. If the domain ontology is well defined, the precision will likely be very high, since all the actors know the terms in the ontology. Another approach is to create the central ontology based on the schemas of the resources, MOMIS uses this method. Using this method could lead to a confusing and not clearly structured ontology. [27]

There are different implementations of using individual (local) ontologies for each information resource or for each set of information resources. SHOE [ref], which is a solution for adding semantics into HTML-files on the WWW or an intranet, uses a base ontology for creating the so called extension ontologies for each information resource. [27]

KRAFT uses mapping between a set of shared ontologies and the resource's local ontology. The mapping is done manual by the ontology author and therefore should be very accurate. This design results in possible disadvantages with specialization losses [27].

Linguistic ontologies are used to simplify the adding of new information, may it be creating a local ontology, mapping between shared and local ontologies or expanding the domain ontology. The linguistic ontology also helps interpreting user queries.Already mentioned KRAFT uses WordNet [28] to define the terms in the shared ontologies. This helps the ontology creators to create the local ontology (if needed) and create the mapping. KRAFT uses all three of these scenarios to obtain a more powerful information retrieval and sharing ability [27].

2.3 Software patterns

There are many problems which are recurring in software development, a structured technique to deal with this is the use of patterns. Software patterns have evolved over the years since Cunningham and Beck introduced their five small guides for novice Smalltalk programmers at OOPSLA'87 [34]. They based their work on the ideas from [29], and adjusted them to object-oriented programming. Patterns became a reoccurring topic at the OOPSLA conferences and a lot of articles and compilations were published.

The so called "Gang of four"; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published their book "Design Patterns" in 94 and it has been very popular for many years. [35] This book became the foundation for many developers, but new ideas keep being introduced, many from [36].

The architect Christopher Alexander formed the idea of patterns in [29] and [30]. The patterns in [29] describe how to do urban planning and how to design buildings. In [30], Alexander describes a pattern as "a three-part rule, which expresses a relation between a certain context, a problem, and a solution". A pattern does not give you the exact solution for the problem, it gives you an explanation how to solve the problem along with the expected result. A good pattern gives insight to the thought process behind the solution; there should be no doubt why the solution was chosen.

2.3.1 Definition

A pattern is an abstract solution to a given problem, in a given context; the pattern defines the relation between the context, problem and a solution. In [31] Software Architecture Patterns are defined as: "*expressing a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.*"

Software patterns include a set of forces for the system, forces can be any attribute that should be taken into consideration, and is affected by the pattern (or in some cases, unaffected). Forces are an important concept to understand when dealing with patterns. The set of forces in a pattern should together form resulting forces which will lead to the solution. This is often referred to as the "tension" that the forces create. Different forces could for example be security and maintainability [32].

Software patterns are usually divided into three groups, architectural, design and idioms. The three groups are defined in [31] as:

- An *architectural pattern* expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

- A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.

- An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

The difference between these different groups is the level of abstraction, where architectural patterns has the highest abstraction level and idioms the lowest.

As described in [33] there are several types of patterns used when designing software systems. The types that will be investigated here are the design patterns and the architecture patterns. In [33] these types of patterns are formally defined with concern of the Intension and locality thesis:

- *Intensional* (vs. *extensional*) specifications are "abstract" in the sense that they can be formally characterized by the use of logic variables that range over an unbounded domain;

- *Non-local* (vs. *local*) specifications are "abstract" in the sense that they pervade *all* parts of the system (as opposed to being limited to some part thereof).

In this paper the authors argue that Architecture patterns are intensional and non-local while design patterns are intensional but local. This means that an architectural description, such as an architecture pattern, must meet both the condition that there are indefinitely many models that satisfy the constraints of the description to meet the intensionallity condition and also that the whole system follows the description and not only parts of it to be non-local. The design patterns however are intensional and local, meaning that there are a infinite number of models that satisfies the description but that the model only describes a part of the system. [33]

2.3.2Pattern languages

Pattern languages should not be confused with a notation to describe a pattern, templates are used to present a pattern. A pattern language is defined by Christopher Alexander as a set of patterns connected to each other, so it can solve a bigger problem in a context, which a single pattern cannot solve. It is not just a collection of patterns, a pattern language includes rules and guidelines suggesting the order to apply each or some of the patterns [37].

2.3.3 Pattern templates

Having a good presentation or notation of a pattern is very important, this makes the pattern easier to read and understand. There are different suggested templates for this purpose. The idea is that the template should facilitate both the creation and use of the patterns. The creation is easier since the template shows which elements that should be present in the pattern and since the patterns will be consistent it is easier to read them. The Hillside community collected a few selected templates by Gang of Four, Doug Lea and AG (AG Communication Systems). Common for all three templates are that they focuses on the relation between the problem, the context and the solution [37].

The Gang of Four introduced their template [38] in their book "Design Patterns", this template uses different expressions to clarify the relation between problem, context and solution; motivation, applicability, intent and structure.

Doug Lea's template [39] has little information in the template regarding what the different headings stands for. For a pattern writer that has a lot of knowledge about how patterns should be written this is not a problem but for a less experienced writer this means that other literature has to be used as a guide.

The template from AG [40] has many sections, and many of them are optional. This gives great possibilities to configure the template to fit the pattern.

2.3.4 Advantages and disadvantages

When developers learn new patterns in projects, they use these patterns in other projects where they are suited. Some developers tend to use patterns where they are not appropriate simply because they are familiar with the pattern, and do not know of any pattern that would fit better. Instead of creating (or inventing) a suitable solution themselves, they implement an inappropriate solution based on a known pattern. [42]

There is a risk that developers will spend valuable time writing patterns that are no benefit to the company or the project, this could be spending time writing patterns for elementary concepts, like linked lists etc., which usually is not needed in a pattern catalogue. Similar menace is when most aspects of the project are expressed as patterns, common example is when existing development techniques are relabeled as patterns without any refinement. This is often referred as pattern overload [43].

Patterns are deliberately underspecified, this allows flexible solutions which are not constrained to application requirements, underlying hardware or OS platform. It is easy to overestimate what patterns actually can do, it is important to have a clear view of what patterns will contribute to the project, and what patterns will not do automatically for the project. In [43] this is called Expectation management.

The advantages of using patterns are that it leads to a common vocabulary for developers; still the vocabularies do not apply to one single programming language. It is easier for developers with different background to share experiences. Patterns help documenting the system architecture; patterns define the different forces, explains why the solution was chosen and which consequences come up, if the chosen pattern is not included in the documentation, this might be lost overtime if not properly documented. This is useful for developing and maintaining the application [42].

In [31] these advantages when using Architectural Software Patterns are stated:

- They help with the recognition of common paradigms, so that high-level relationships between software systems can be understood and new applications built as variations on old systems.
- They provide support for finding an appropriate architecture for the software system under development.
- They provide support for making principled choices among design alternatives.
- They help with the analysis and description of high-level properties of complex software systems.
- They provide support for change and evolution of software systems.

2.3.5 Patternity tests

Patterns that are "waiting" and not yet documented are often called "proto-patterns". To be able to distinguish what really constitutes a good pattern, patternity tests can be used. These tests are usually of the type pass/fail and have a number of criteria's that the pattern must pass or it is probably not a pattern. The patternity tests are derived from a pattern definition like the one on [41] and these examples are from [44] and tests the following criteria of the practice that may constitute a pattern:

Proven Practice: It is tried and true, time-tested and battle-proven practice that has been used three times at least.

Geometry/Structure: It should be concerned with some kind of structure that can be visually depicted.

Forces: It must have forces, concerns, goals, constraints, trade-offs, etc.

Concreteness: It should solve a concrete problem in a concrete context.

Generative: It should show the reader how to create something and what is to be created.

Enhances Quality of Life: A pattern solution needs to demonstrate how/why it is desirable. It should show how it adds value to the architecture and/or to the lives of its users and practitioners and developers.

Peer Validation: If something is a genuinely recurring practice employed by expert practitioners, then one's practicing expert peers should be able to validate it as such. Find several people you know that are very knowledgeable about the problem domain and ask if they recognize the basic form of the solution you have in mind. If most of them do not, you may need to find some more abundant or convincing examples of Proven Practice before you can call it a pattern.

These patternity tests comply well with the definition of good patterns made in [41]. The difference is that these tests are better defined with more granularity than using the definition of a good pattern on [41].

2.3.6 Ontology patterns

The use of patterns in the ontology community has not been widely adopted and the few patterns that have been constructed are generally language specific or specialized for a specific type of ontology. In [45] five levels of ontology patterns are described, where the syntactical patterns are at the lowest level of abstraction and the application patterns are at the highest level:

- Syntactic patterns, theses patterns are language specific and are used to arrange representation symbols in order to create a certain concept, relation or axiom [45].

- Semantic patterns, patterns that are a meta-description of a Syntactic pattern, i.e. patterns that are language independent descriptions of a certain concept, relation or axiom [45].

- Design patterns, these are patterns that are used to form a small piece of a complete ontology. The patterns are a small collection of Semantic patterns [45].

- Architecture patterns, a collection of Design patterns that together describe the structure of the complete ontology [45].

- Applications patterns, these are patterns that aim to describe the generic ways of implementing the constructed ontologies, i.e. the purpose, scope, usage and context of the implemented ontology or ontologies, including interfaces and relations to other system [45].

2.4 Software architectures

A description of a software architecture is the description of the high hierarchical structure of a software system. This description describes the overall design of the system that includes global control structure, communication protocol, data access and the systems major components and the behavior of the components [46].

The essential idea of this description is abstraction, to hide some of the details of the system in order to make it easier to understand the properties of the system. If the system is complex there can be several levels of abstraction, and the elements in each level can be decomposed into new architectures [46].

The configurations of the present elements in a software are what defines the architecture. These elements can be divided into components, connectors and data and the configuration is the relationship between these elements. The component is an abstract unit of software instructions and internal state that provides a transformation of data. This can be transformations like computation or loading data to memory from secondary storage. Components are the major element of a software architecture. A component can be large grained and provide lots of functionality, this is typically a subsystem, or fine grained like a function [46].

The connectors are an abstract mechanism that provides the ability of communication between the components. A connector may have a subsystem of components inside in order to make this communication possible and this can be modeled on a lower level of abstraction. On a higher level this is not modeled since the connector is used to hide this complexity in the model [46].

Data is the information that is transferred between the components in the architecture. A single entity of this type is called a datum. The nature of these elements in an architecture are for example essential in network-based application since this often determines if a certain architectural style will be appropriate [46].

An architecture style describes the organization structure of the software system which means that it describes how components and connectors can be configured into a software system. If an architecture style is more formally described they constitute architecture patterns, such as the patterns in [31]. Examples of architecture styles are Pipe and Filter where the components are filters while the connectors are the pipes. The components takes data from the inputs and produces outputs the connectors that transmits the data to other components. This gives the benefit that the architect of the system can view the whole the system as a combination of filters. Other types of architecture styles are the Data Abstraction and Object oriented Structure. This is the object oriented structure which is very common, the components encapsulate the data and the operations and the connectors make it possible for the components, or objects as they are called in this structure, to communicate. The connectors are themselves procedures. Other styles are the event based Implicit Invocation where events trigger the components and the layered system where each layer in the system provides services to the layer above it and the layers can only communicate whit the layer directly above or below them. This is a very common structure for example in database management systems [46].

2.5 Architecture Description Languages

In order to support the construction of software architectures the use of architecture description languages (ADLs) have been proposed. An ADL can be defined as “an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module” [48]. ADLs are divided into software and hardware ADLs. Software ADLs are used to describe software systems and the description and the components in the language are software processes or modules and the ADL is used to define and model the software architecture. The hardware ADLs describe the hardware components in a system, and a usuall application for such languages are systems with application specific instruction-set processors[47], and in this case the processors are describes in terms of their instruction sets.

There are several ADLs and examples of these are Rapide, UniCon, and Darwin. In order to make it possible to transfer an architecture description from one ADL to another the ADL ACME has been constructed to make it possible to map one ADL to another [48].

Another ADL is xADL, which has been developed by the University of California, Irvine and is defined as a set of XML schemas. The language has a core model with four elements: Components, Connectors, Interfaces and Configurations. The first three are the same as described in 2.5 and the Configurations are the topological arrangements of components and connectors as realized by links. The language has a modular design which makes it easy to extend with new structures. For a more thorough description of the language see [49].

3 Methodology

The method we will use is divided in two parts, to find an definition of ontology application patterns and examine what is needed to describe them and the other part to find benefits and problems with using these types of patterns. For both of these methods the data needed will be collected through literature studies.

The general idea regarding the first research question is to study other types of software patterns and see if any conclusions can be drawn to whether the ontology patterns resemble other types of patterns. If such conclusions can be made, a definition could be formed using the definitions of other types of patterns.

To find benefits and problems with using patterns, the method we will use is to research the benefits and problems of other types of patterns and how these types of patterns relate to the patterns we will construct. If we find pattern types that are closely related, the benefits and problems are perhaps also closely related as well. This research will be conducted through extensive literature research and the results of this research will be the base for a discussion around the benefits and problems of ontology application patterns.

4 Realization

In order to find what really constitute an ontology application pattern a good starting point could be to look at other type of patterns and see how they might relate to the ontology application patterns.

To see how this relates to ontology patterns one could try to see if the ontology patterns also fulfill the intensional and locality conditions described in 2.3.1 in order to see if there is a resemblance. So this level of ontology patterns are used to describe the use of whole ontologies or the use of systems of ontologies. The intensionality condition is broken if these types of patterns only can be used in less than infinite number of solutions and the locality condition is broken if the patterns do not describe a whole system.

So with the description of Ontology Application Patterns described in 2.3.6, what can be said concerning ontology application patterns regarding this issue? Since the patterns give a solution that includes the use of one or more ontologies which have some sort of relation to outside systems, it is reasonable to say that it is a whole system that is modeled, not only parts of systems. This should mean that these patterns would be non-local.

If these patterns were to be not intensional, there must be a finite number of models that can be created when using the patterns. Since these patterns are high-level patterns they only constrain what have to be constrained, and leave the rest up to the person using the pattern. This gives the possibility to create an infinite amount of models which programs can be constructed from. This gives that these types of patterns are intensional. To conclude, when looking on the description of these types of patterns from 2.3.6, ontology application patterns have a lot in common with software architecture pattern since they are both intensional and non-local. This is also the characteristics of the software architecture patterns. This could mean that these types of patterns are related or at least have a lot in common. However, since Ontology Application Patterns are new and little research has been made in this area, good examples of these patterns are very hard to find. This makes it hard to really verify if the intesionality and locality criteria really holds up since a formal description of non-existing patterns are hard to do.

4.1 Ontology application pattern definition

Since ontology application patterns are very much related to Software Architecture Patterns, the definition of out patterns should reflect that, although ontology application patterns include some sort of ontology.

Our definition of ontology application pattern will therefore be a combination of these two definitions where the definition of Software Architecture Patterns as written in 2.3.1 is used to enhance the description described in 2.3.6. Our definition is:

The fundamental structural organization schema for the software system that the pattern describes. This software system utilizes ontologies in some way to create some functionality. The pattern provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. It also describes properties of the ontology or ontologies in the system and the connection between the ontology and the rest of the system.

4.2 Ontology properties

The difference between the software architecture patterns and the ontology application patterns are that there always exist ontologies in the latter and that the pattern should give insight in what capabilities this ontology should have. This leads to the question, what characteristics should be described concerning a whole ontology in an ontology application pattern? In 2.1.3 examples of these characteristics are given. These dimensions creates conditions on the ontology described so the reader of the pattern can understand what demands that are put on the ontology to make the solution work.

In order to make it easier to create ontology Application Patterns one of the tasks in this thesis is to investigate whether it is possible to create a template of what these patterns should include and how to best describe them.

Software patterns are usually documented using a pattern template as described in 2.3.3, and this is also reasonable in this case. The big difference between software architecture patterns and the ontology application patterns are the presence of ontologies in the latter.

Software architecture patterns are often described using an Architecture Description Language and these languages usually contain the elements components and connectors. If ontologies are treated as another component there are several issues to be considered, like what are an ontologies interfaces and what has to be described concerning the ontology are examples of these issues. If the component description in an ADL were to be extended by the characteristics of the ontology this could be a way to show what the ontologies must be able to do within the system.

As described in section 2.5 there are several ADLs available. In order to make the work of extending an ADL to accommodate the use of ontologies an ADL with the property of being very extendable were choosen, namely xADL 2.0.

xADL is highly extensible and has a modular organization, where new modules can be added to give more functionality in the language. This is exactly what might be a good way to create a description of a system using ontologies i.e. creating an ADL for ontology applications.

Ontologies can be seen as a component in the system with some sort of information inside and given interfaces to the ontology, other components within the system should be able to access and manipulate the ontology. With this in mind, a component with the properties of an ontology should be added to the language. The interface types also have to be defined and connectors has to be described.

The mentioned dimensions in 4.2 are used as the properties of the ontology and a new element called ontology is created in xADL. In order to make it possible to describe the ontology and what dimensions it should have to be able to fulfill the purpose in the pattern some sort of description is necessary. The ontology element gets a description called the Ontology description which has all the ontology dimensions as attributes. The structure of the Ontology element can be seen in Figure 1.

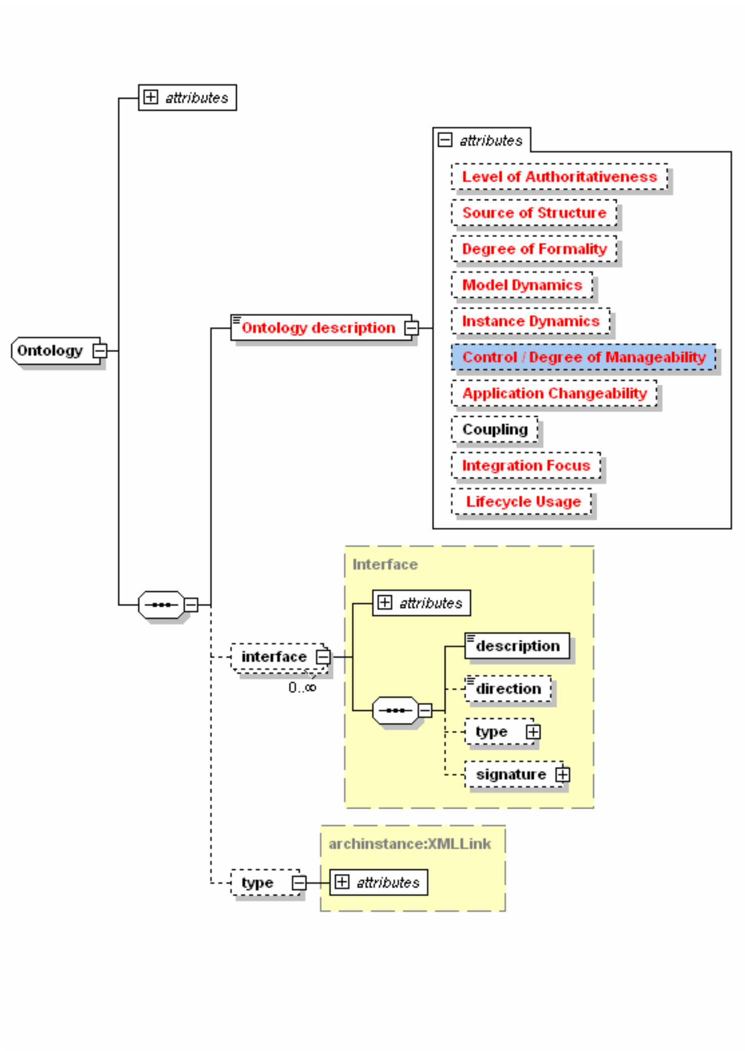


Figure 1 The Ontology Element in xADL

The interfaces are the connection points to the ontology and should describe what services the ontology can give the other components. In 2.1.4 examples of such services are given and those are examples of what can be considered ontology interfaces but the problematic part is that this might be an abstraction too high and these are interfaces provided by components with more intelligence such as inference engines. However this does not leave a lot of services that the ontology can perform on its own. I think that the abstraction level of the description should decide what the interface should be. If the abstraction of the system is high, the ontology as a component includes a lot of intelligence such as inference engine and the interface will show very advanced services that this component can do. If on the other hand the abstraction level is low, this component is divided into several subcomponents and the ontology's interface should be more of the read and write kind.

The last of the added elements is the connector. This is the element that makes the communication with the ontologies possible, but what should a connector consist of? When examining possible patterns like the use of ontologies in knowledge management systems, or in multi agent systems such as KRAFT, or InfoSleuth the connections between ontologies is best described as mappings between the ontologies. This means that a connector could be mapping-information and be a rather large element with lots of information. The connector could also be a simpler element where a component such as a query engine fetches information from the ontology. This means that there are two types of connectors, connectors between ontologies and connectors which connect a standard component to an ontology.

4.3 Definition of pattern template

We have chosen the pattern template from AG, since the good support and flexibility of this template makes it the most suitable we could find.

The template and short descriptions of each topic will follow. This should not be considered as a strict line of rules; each topic is not always needed for a good pattern. All references to pattern names should be written in italic.

[Name of pattern]
Aliases: ***Aliases*** (If there are any)

Problem
Gives a short description of the problem. This could be a statement or a question.
Context
The context in which the pattern is valid.
Forces
Lists and describes each relevant force.

- Force one
- Force two
- Force …

Solution

Gives a statement of the solution to the problem, can include diagrams.
Resulting Context
Describes the context after the solution is applied.
Rationale
Explains the logic behind the solution. The user will understand why this solution is a good solution.
Known Uses
Gives examples on where the pattern can be used.
Related Patterns
Lists any related patterns.
Sketch
Describes the sketch, if needed.

Author(s): Names of the authors.
Date: Date
Author's emails
Pattern Source: The source of the pattern, for example University of Jönköping, IT research department.

References
Gives a list of references cited in the pattern.

Keywords: A comma delimited string of terms used for searching.

Example

This could be pseudo code, skeleton for classes etc.

4.4 Applicability, benefits and issues of ontology application patterns

As already stated Ontology Application Patterns are closely related to architectural software patterns and this could mean that the benefits and issues are also related. The problem is that I have not found a lot of information concerning the experience of using explicitly architectural software patterns other than the advantages stated in 2.3.4. But since I think that the same benefits and issues probably are present if Ontology Application Patterns where to be used, I have examined those advantages and issues and have reasoned whether they may occur.

Architectural patterns are best applied at the beginning of the architectural design when the structure of the application is designed, and we think that this holds true for our pattern suggestions also. Ontology Application Patterns are high-level patterns that show the fundamental structure of an application, not how the more fine-grained modules of the application should be implemented.

The main difference between ontology application patterns and architectural software patterns are that the ontology application patterns specifically state the properties of the ontology within the solution. But the goal of giving an insight of the impact of choosing a specific structure of the application is similar. Our opinion is that the potential benefits of using architectural patterns also apply to Ontology Application Patterns. These benefits are stated in 2.3.4. but we will repeat them here and give a motivation why they should apply. The strengths of applying architectural patterns are written in italic and our motivation is written in normal font:

They help with the recognition of common paradigms, so that high-level relationships between software systems can be understood and new applications built as variations on old systems.

This should be true since Ontology Application Patterns also build on common paradigms in existing systems.

They provide support for finding an appropriate architecture for the software system under development.

Since Ontology Application Patterns show architectures and the reasons for using them this should apply.

They provide support for making principled choices among design alternatives.

Ontology Application Patterns can be used to show alternative solutions to similar problems, specifically the different organization of the ontologies in the information integration patterns.

They help with the analysis and description of high-level properties of complex software systems.

Ontology Application Patterns try to give an insight into complex systems, so this should apply.

They provide support for change and evolution of software systems.

The idea of the Ontology Application Patterns is to give an insight to not only how an architecture is designed but also why, thus making it easier to understand the system in the future and what principles that where used in the design. This should also apply.

The problem here is that I have no empirical experience to base these assumptions on.

Also the advantages stated to using design patterns should apply to using ontology application patterns. I have examined the experience reports mentioned in 2.3.4. to find what weaknesses and strengths design patterns has and examine if they might also apply on ontology application patterns. I think that the strength of using patterns as a sort of common vocabulary should apply, provided that the naming of the Ontology Application Patterns are good and the quality of the patterns themselves are good. If they are not used the effect of the common vocabulary also disappears. The rest of the benefits mentioned in those experience reports are rather redundant and already included in the mentioned benefits.

The weaknesses of Ontology Application Patterns are of the generic type described in 2.3.4, such as developers using inappropriate patterns and overestimation of what the contribution of the patterns might be.

5 Results and discussion

Two problem formulations were the basis for this thesis, to examine if it is possible to find a definition of high-level patterns of ontology based applications and examine how these patterns relate to other types of software patterns is the first one. The second one was to investigate the strengths and weaknesses of these patterns and how they could be used.

An examination of the relationships between ontology application patterns and software architecture patterns have showed that these are related with one difference, the ontology application patterns include at least one ontology. This means that there are demands on the pattern description that are not present in standard architecture patterns. The properties of the ontologies has to be described and the connections to the ontologies also has to be given properties.

This reasoning has led to a definition of Ontology Application Patterns has been made in 4.1. To be able to document in Ontology Application Patterns a template is needed and in this thesis a suggestion of such template has been made in 4.3. The template was chosen because of its flexibility and an example of how xADL could be extended to be able to describe ontologies has been constructed in 4.2.

Finding the answer to the second research question proved to be difficult. The answer is based on previous experience from other related types of patterns, since there really are no experiences from Ontology Application Patterns available. The close relation between ontology application patterns and architectural software patterns made it possible to finds some possible strengths and weaknesses that the use of these patterns suggestions could have. The same relation makes it very likely that ontology application patterns should be applied in an early stage of the development process.

6 References

[1] Guarino, N. and Poli, R. (eds.) 1995. "Formal Ontology in Conceptual Analysis and Knowledge Representation," *Special issue of the International Journal of Human and Computer Studies*, vol. 43 n. 5/6, Academic Press.

[2] Gruber, T. . *"*A translation approach to portable ontology specification,*" KnowledgeAcquisition* 5:199-220; 1993

[3] M. Uschold and M. Gruninger, "Ontologies: principles, methods, and applications", *Knowledge Engineering Review*, 11(2), 93--155, (1996).

[4] Pease, A., Niles, I., and Li, J. 2002. The Suggested Upper Merged Ontology: A Large Ontology for the Semantic Web and its Applications. In *Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web*, Edmonton, Canada, July 28-August 1, 2002.

[5] Mike Uschold, Martin King, Stuart Moralee and Yannis Zorgios , "The Enterprise Ontology", *The Knowledge Engineering Review* , Vol. 13, Special Issue on Putting Ontologies to Use (eds. Mike Uschold and Austin Tate).
Also available from AIAI as AIAI-TR-195:

[6] D. Lenat and R. Guha, *Building large knowledge-based systems: representation and inference in the cyc project*, 1990. Addison-Wesley, Reading, Massachusetts.

[7] A. Farquhar, R. Fikes, and J. Rice, "The ontolingua server: A tool for collaborative ontology construction", *In Proceeding of the 10th Knowledge Acquisition for Knowledge-based Systems workshop,* 1996.

[8] M. Kifer, G. Lausen, and J. Wu, „Logical foundations of object oriented and frame-based languages", *Journal of the ACM, 1995.*

[9] T. Berners-Lee. The semantic web. *Scientific american*, 284(5):35–43, 2001.

[10] *http://www.w3.org/TR/rdf-primer/* (2006-02-16)

[11] *http://www.w3.org/TR/rdf-schema/* (2006-02-16)

[12] Fensel, D. et al, "OIL in a nutshell", in *R. Dieng et al. (eds.), Knowledge Acquisition, Modeling, and Lanagement, Proceedings of the European Knowledge Acquisition Conference* (EKAW-2000).

[13] "OWL Web Ontology reference" *http://www.w3.org/TR/owl-ref/* (2006-02-16)

[14] M. Klein. "Combining and relating ontologies: an analysis of problems and solutions", *Proceedings of the IJCAI-01 Workshop on Ontologies and Information Sharing Seattle, USA, August 4-5, 2001.*

[15] Niles, I and Pease, A. (2001) "Origins of the IEEE Standard Upper Ontology," *Working Notes of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology,* Seattle, 3742.

[16] A. D. Preece, K.-Y. Hui, W. A. Gray, P. Marti, T. J. M. Bench-Capon, D. M. Jones, and Z. Cui, “The KRAFT Architecture for Knowledge Fusion and Transformation*”,* in *19th SGES Int. Conf. on Knowledge-based Systems and Applied Articial Intelligence (ES'99).* SpringerVerlag, 1999

[17] Luc Steels and Paul Vogt, “Grounding adaptive language games in robotic agents”, in *C. Husbands and I. Harvey, editors, Proceedings of the Fourth European Conference on Artificial Life*, Cambridge MA and London, 1997. The MIT Press

[18] “Ontology Definition Metamodel*, “* *http://www.omg.org/docs/ad/06-05-01.pdf* (20060810)

[19] O. Dameron, N. F. Noy, H. Knublauch, M. A. Musen, “Accessing and Manipulating Ontologies Using Web Services” ,*Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications at the Third International Conference on the Semantic Web* (ISWC-2004), Hisroshima, Japan

[20] Jasper, R. and Uschold, M. 1999. “A Framework for Understanding and Classifying Ontology Applications”, *In Twelfth Workshop on Knowledge Acquisition Modeling and Management KAW'99*

[21] OntoWeb, D21, Successful Scenarios for Ontology-based Applications

[22] R. J. Bayardo Jr., W. Bohrer, and et. al., “InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments”*, ACM SIGMOD, 1997.*

[23] Cui Z, Jones D, O'Brien P, “Semantic B2B integration: issues in ontology-based approaches”, *SIGMOD Rec* 31(1): 43—48

[24] Fox, M., & Gruninger, M, “On Ontologies and Enterprise Modelling”, in *Proceedings of International Conference on Enterprise Integration Modelling Technology 97*. Springer-Verlag.

[25] S. Bergamaschi, S. Castano e M. Vincini "Semantic Integration of Semistructured and Structured Data Sources", *SIGMOD Record Special Issue on Semantic Interoperability in Global Information*, Vol. 28, No. 1, March 1999

[26] Deschaine L.M., Brice R., Nodine M, “Use of InfoSleuth to Coordinate Information Acquisition, Tracking and Analysis in Complex Applications”, *Tech. report MCC.*

[27] Zyl J., Corbett D, “Population of a Framework for Understanding and Classifying Ontology Applications”, *citeseer.ist.psu.edu/442857.html* (060913)

[28] *http://wordnet.princeton.edu/* (2006-01-21)

[29] C. Alexander*, A Pattern Language*, Oxford University Press (1977).

[30] C. Alexander, *A Timeless Way Of Building*, Oxford University Press (1979).

[31] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *PatternOriented Software Architecture -- A System of Patterns*. Wiley, 1996.

[32] J.O. Coplien, *Software Patterns: A Pattern Definition, Bell Laboratories,* HillSide

[33] Amnon H. Eden Rick Kazman, "Architecture, Design, Implementation", in *International Conference on Software Engineering – ICSE*, May 3-10, 2003, Portland, OR

[34] K. Beck and W. Cunningham, *Using Pattern Languages for Object-Oriented Programs*, OOPSLA-87, Technical Report No. CR-87-43, September 17, 1987

[35] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, Massachusetts, January 1995, ISBN 0201633612.

[36] *http://www.hillside.net (*2006-09-13)

[37] B. Appleton, Patterns and Software: *Essential Concepts and Terminology (Feb 2000)*

[38] *http://hillside.net/patterns/writing/GOFtemplate.htm (*2006-09-13)

[39] *http://hillside.net/patterns/writing/Lea.htm (*2006-09-13)

[40] *http://hillside.net/patterns/writing/ags_pattern_template.htm (*2006-09-13)

[41] http://hillside.net/patterns/definition.html (2006-09-13)

[42] D. C. Schmidt, "ExperienceUsingDesign Patterns to Develop Reuseable Object-Oriented Communication Software," *Communicationsof the ACM (Special Issue on Object-Oriented Experiences)*, vol. 38, October 1995.

[43] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9th *European ConferenceonObject-Oriented Programming*, (Aarhus,Denmark),ACM, August 1995.

[44] *http://c2.com/cgi/wiki?PatternityTests* (2006-09-13)

[45] E Blomqvist, K Sandkuhl. "*Patterns in ontology engineering: Classification of ontology patterns".*

[46] Roy T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Ph.D. Thesis, University of California, Irvine, Irvine, California, 2000.

[47] Ravi Saini, Pramod Tanwar, A. S. Mandal, S. C. Bose, Raj Singh, Chandra Shekhar, "Design of an Application Specific Instruction Set Processor for Parametric Speech Synthesis," *vlsid*, p. 773, 17th International Conference on VLSI Design, 2004.

[48] Nenad Medvidovic, Richard N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93, Jan., 2000.

[49] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor “A Highly-Extensible, XML-Based Architecture Description Language”
In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam Netherlands.