Development of an ontology for the problem space of architectural design

Wouter Pinnoo

Supervisor: Prof. dr. ir. Frank Gielen
Counsellors: Ir. Jolien Coenraets, Ir. Jelle Nelis

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Department of Information Technology
Chair: Prof. dr. ir. Daniël De Zutter
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT
GENT

Development of an ontology for the problem space of architectural design

Wouter Pinnoo

Supervisor: Prof. dr. ir. Frank Gielen
Counsellors: Ir. Jolien Coenraets, Ir. Jelle Nelis

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Department of Information Technology
Chair: Prof. dr. ir. Daniël De Zutter
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

Preface

In my first year in the master Computer Science Engineering, I had the opportunity to attend the course *Software architectures*, taught by prof. dr. ir. Frank Gielen, which aroused my interest in this domain of software engineering that was completely new to me. In the selection process of master dissertation subjects, it was obvious for me to pick this subject as first choice since it focuses on a practical problem in the domain of software architectures.

Without doubt, the result of this master dissertation would not have been achieved without the help of my mentors at the university. I would like to thank my supervisor prof. dr. ir. Frank Gielen for giving me the opportunity to work on this subject and for his assistance. I also would like to thank my mentors ir. Jolien Coenraets and ir. Jelle Nelis for their insights, guidance and a lot of practical assistance throughout the year.

Furthermore, I would like to thank dr. Femke Ongenae for her assistance with ontologies and dr. ir. Koen Yskout and dr. ir. Dimitri Van Landuyt for sharing their expertise and experiences on practical use of software architectures, and the guidance on making the result of this thesis useful for software architects.

Finally, I thank my parents, family and friends for their support throughout my studies at Ghent University. I also owe my greatest gratitude to my girlfriend Celine for supporting me during the time-consuming process of making this master dissertation.

Ghent, June 2016
Wouter Pinnoo

Permission for usage

Ghent, June 2016
Wouter Pinnoo

Development of an ontology for the problem space of architectural design

by Wouter Pinnoo
Academic year 2015-2016

Ghent University
Faculty of Engineering and Architecture
Department of Information Technology
Chair: Prof. dr. ir. Daniël De Zutter
Supervisor: Prof. dr. ir. Frank Gielen
Counsellors: Ir. Jolien Coenraets, Ir. Jelle Nelis

This paper is based on a master dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering in June 2016.

Abstract

Since many problems and requirements in the domain of software architecture recur, software architects make use of documented patterns that describe solutions to common problems. For efficiency reasons, it is in the architect's best interest that these patterns are documented in a consistent way and are easily retrievable. This master dissertation focuses on firstly researching the different kinds of classification methods of such patterns, and secondly on developing an ontology model that can capture all the needed pattern information to use for classification. With this ontology as a basis, a tool is made that can aid software architects in finding relevant architectural patterns without explicit knowledge of the ontology. An elaborate introduction is given in Chapter 1, along with related work. In Chapter 2, a classification is made of a selected set of architectural patterns, in order to get insights of which properties of a pattern are most rewarding for classification. With these classification insights in mind, Chapter 3 describes the design of the ontology model, followed by a discussion on querying this ontology in Chapter 4. In Chapter 5, the design of a querying tool is described that aids architects in querying the ontology. Its architecture and related technical decisions are discussed in Chapter 6. An evaluation and self-reflection is made in Chapter 7. Finally, this master dissertation concludes with a summary and gives suggestions on future work in Chapter 8.

Keywords: ontology, software architecture, pattern

Development of an ontology for the problem space of architectural design

Wouter Pinnoo
Supervisor: Prof. dr. ir. Frank Gielen
Counsellors: Ir. Jolien Coenraets, Ir. Jelle Nelis

***Abstract*—Software architecture is a very broad domain in the field of software engineering, and the practitioners — software architects — often don't find their way through the scattered architecture documentation. Knowledge about architectural patterns is by far the most important asset of an architect, and having a more coherent way to find pattern information can improve the architect's efficiency.**

This paper is based on a master dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering at Ghent University in June 2016. This master dissertation presents an ontology of architectural patterns (with their related concepts) that captures pattern knowledge in a consistent way and allows to retrieve related pattern information with queries on the ontology. In addition, a prototype of a web-based utility tool is made to illustrate the practical possibilities of the ontology while hiding the technical complexity for the end user.

***Index Terms*—ontology, software architecture, pattern**

I. Introduction

SOFTWARE architectures are the building blocks for every software application. To make optimal use of already acquired knowledge about real-world problems, software architects make use of architectural patterns. Architectural patterns define, given a certain environment and some constraints, how a problem can be tackled, and how to implement and document it. Since pattern documentation is very scattered and the definitions of concepts related to patterns are not consistent in literature, it is difficult for architects to find their way in the literature.

In this master dissertation, existing pattern classification methods were researched to form the basis for an ontology that captures all information related to patterns, as will be discussed in Section II and III respectively. Section IV presents a web-based tool that can be used by architects to search for patterns, and retrieve information about the patterns and related concepts. Next, an evaluation of the ontology and web-tool is made in Section V. Finally, this paper concludes in Section VI.

II. Pattern classification

In some of the most used books about architectural patterns, and software architectures in general, several existing pattern classifications can be found. Unfortunately, most of them are not compatible with each other, but are rather only a classification of the relative small set of patterns described by the author. However, insights from all those pattern classification methods can be used to construct a more general categorisation of patterns.

Some authors make use of *viewtypes* to categorise patterns, others use the *purposes* of the patterns to classify the patterns. Yet others distinguish patterns based on their internal *structure*. A classification schema was developed that captures all of the researched information. The classification schema generalises categorisation methods found in following resources:

- *Software Architecture in Practice* (Bass et al. [2])
- *Documenting software architectures: views and beyond* (Clements et al. [5])
- *Architectural Patterns Revisited – A Pattern Language* (Avgeriou et al. [1])
- *Microsoft Application Architecture Guide* (Microsoft Patterns & Practices Team [7])
- *Pattern Oriented Software Architecture* (Buschmann et al. [4])
- *Software Architecture: Perspectives on an Emerging Discipline* (Shaw et al. [8])

Although the developed categorisation schema generalises information found in literature, it still maintains the authentic relationships defined in literature and keeps track of which resource — and optionally which page or section in the resource — defines the relationship. With *relationship* can be understood: *any* relationship between *any* object discussed in literature. For example, both relationships between two patterns, and between a pattern and a viewtype are considered.

III. Ontology

Once the classification schema was finished, an ontology was built using the categorisation insights gained from the schema.

A. Structure

On Fig. 1, a depiction of the concepts of the developed ontology model is made. Next to a *Pattern* class, the ontology can comprise all objects that are related to a pattern and can serve as a categorisation method. For example, from the different existing categorisation methods found in literature, *Viewtypes* were found to be useful categories. Also, *Quality Attributes* were proven to be very useful parameters for architects when searching patterns. Bass et al. [2], for example, categorise all patterns in their book based on the quality attributes that are relevant for the patterns.

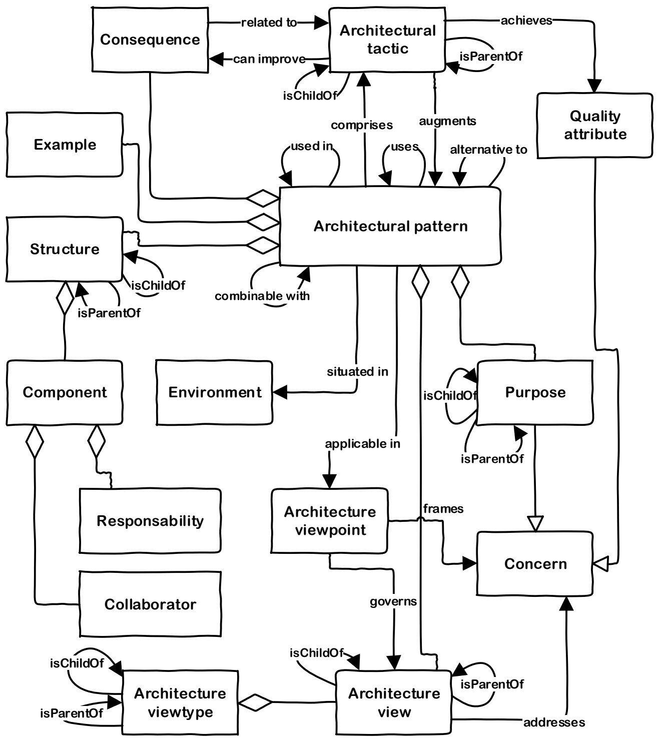


Fig. 1: Ontology model

All such concepts were captured in the ontology and compared to the concept definitions found in the ISO-42010 standard [6], so that definitions from this ontology comply with the ISO standard.

Besides the inclusion of concepts related to architectural patterns in the ontology, the ontology is also developed to contain references to corresponding resources, or plain-text annotations. This way, all relationships from the designed classification schema can be included in the ontology. Also, each relationship can be annotated with the literature resource that defined the relationship. This allows the user to consult relevant external literature resources when interested in a relationship found in the ontology. A datatype specific for DBpedia was added to the ontology, so that objects can be annotated with DBpedia ontology URLs. DBpedia is a publicly available ontology that contains information extracted from Wikipedia [3] and is thus a good extension of the patterns ontology: abstracts found on Wikipedia are annotated to concepts in the patterns ontology to provide end users with short descriptions of the concepts.

B. Queries

The design of the ontology model was made with ease of querying in mind. The ontology will be used for querying patterns — and related information — based on some categorisation parameters. As a result, inheritance of objects was chosen to be implemented using object properties rather than ontology class inheritance. The reason for this, is that it is easier to perform complicated queries on OWL individuals than on classes.

SPARQL engines with HTTP interfaces were used to query the ontology with SPARQL queries via HTTP calls. Several engines were tested and compared in terms of execution time, ease of use and supported features.

IV. Web-based tool

To overcome the problem of software architects having issues with finding the architectural pattern information they need in an efficient way, a web-based utility tool was developed. With this tool, architects are not required to have the knowledge about ontology querying languages to be able to use the developed ontology. The tool hides the complexity of the ontology queries and allows the end users to retrieve pattern information using a more generalised categorisation method. Also, the end user can easily view information about patterns found in different literature resources and find references to the resources.

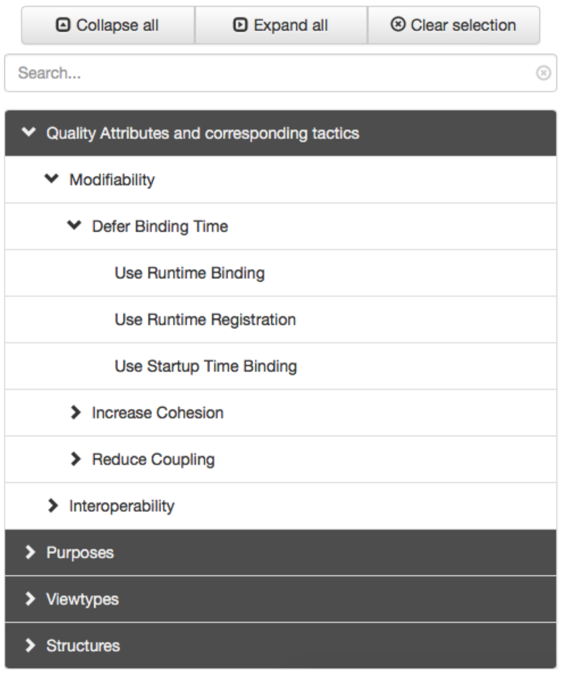


Fig. 2: Screenshot of categories in the web tool

Fig. 2 shows a screenshot of the part of the homepage of the web tool where the pattern categories are listed. This list of categories was retrieved from the ontology. Firstly, it contains the quality attributes, with all tactics that achieve the quality attributes as child nodes; followed by all purposes of the patterns, viewtypes related to them, and the internal structures of patterns. Next, the user can select one or more items in this category list, after which the tool will query the ontology for relevant patterns.

In addition, the web tool provides a catalog of all the objects contained in the ontology. When selecting an item, the user is redirected to a page that contains all the details of the selected object. These details are gathered using queries that retrieve all the information from the ontology that is related to the object, along with some additional data. For example, when querying details of an architectural pattern, the page will show a Wikipedia abstract of the pattern, a list of resources that discuss details of the pattern, and relationships to *Purposes*, *Views*, *Viewtypes*, etc. All relationship annotations are also shown — this allows the end user to find relevant literature resources.

Finally, the web tool contains a hierarchical visualisation of all categorisation methods. The user can browse through patterns belonging to each categorisation method (*Purposes*, *Quality Attributes*, etc.) and further browse through concepts related to the pattern. This page allows to visualise relationships without having to deal with (relatively large) loading times of other pages.

V. Evaluation

After developing the ontology and a web-based utility tool for the retrieval of architectural pattern information, an evaluation and self-reflection is made of the end product of this master dissertation, with gives suggestions to future work initiatives.

A. Ontology

Since the ontology was developed to be generic enough to be able to capture new information, new literature sources were compared against the ontology model and a conclusion was made on to which extent they fit in the existing ontology model. The main conclusion was that most of the information found in new literature sources fit in the ontology with the exception of some details in the templates that were used to structure pattern information in literature. Some of this information was not concrete enough to fit in the ontology but can be included in the form of plain-text annotations.

Also, a conclusion was made that the separation between theoretical pattern documentation and documentation of already implemented patterns is very important and should be respected when adding content to the ontology. For example, documentation of patterns that are already implemented have extensive descriptions of the views of the patterns, while these are usually not know in advance — theoretical pattern documentation only contains prescriptions on how to use the pattern. This suggests that a.o. some modifications are needed to the conceptualisation of the internal structure of a pattern in the ontology.

B. Web-based tool

As the goal of the web-based utility tool was to provide the end user with visualisations that hide technical complexity of the ontology, and to increase the efficiency of retrieval of pattern information, some use cases were developed. A manual of the tool was given to five participants and they were asked to execute the steps defined in these use cases, while their interaction with the tool was observed.

Most of the gained feedback was related to minor visual improvements that could be made in the tool, but in the majority of use cases, the expected results were successfully achieved by the participants. Although not all users used the same visualisation pages to achieve the desired result, a conclusion was made that the participants were able to efficiently retrieve the needed information. Some feedback suggested that the tool will be even more user-friendly if it contains more descriptive information about the objects. Also, more explanation in the manual about the used relationships in the ontology would have helped the participants.

VI. Conclusion

Since efficient retrieval of relevant architectural patterns helps software architects, research of existing classification methods was performed in this master dissertation. An ontology of architectural patterns and their related concepts was made, complying with the ISO-42010 [6] standard, to eliminate the issue of incompatibility and inconsistency of pattern documentation across literature. Class instances and relationships in this ontology can be annotated with resource links, so that the end users can easily find relevant literature resources. Also, Wikipedia abstracts can be included in annotations in the form of DBpedia ontology URLs. Plain-text information can be used to annotate objects with short descriptions.

A web-based tool was developed to aid architects in the retrieval of relevant information, and to hide the complexity of queries on the ontology. The ontology, along with examples of queries and the prototype of the web tool, form a solid base for extensions in future work that can cover even more literature sources or allow the ontology to contain other pattern-related information.

References

[1] Avgeriou, P. and Zdun, U. *Architectural Patterns Revisited - A Pattern Language*. Information Systems Journal, 81:1–39, 2005. doi: 10.1.1.141.7444.

[2] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, volume 2. 2003. ISBN 0321154959.

[3] Bizer, C., Cyganiak, R., Auer, S., and Kobilarov, G. *DBpedia.org—Querying Wikipedia like a Database*. In Developers track at 16th International World Wide Web Conference (WWW2007), Banff, May 2007.

[4] Buschmann, F., Henney, K., and Schmidt, D. C. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, volume 5. Wiley Publishing, 2007. ISBN 0471486485.

[5] Clements, P., Garlan, D., Little, R., Nord, R., and Stafford, J. *Documenting software architectures: views and beyond*. 25th International Conference on Software Engineering, 2003. Proceedings., pages 3–4, 2003. ISSN 0270-5257. doi: 10.1109/ICSE.2003.1201264.

[6] International Organization Of Standardization. *ISO/IEC/IEEE 42010:2011 - Systems and software engineering – Architecture description*. ISOIECIEEE 420102011E Revision of ISOIEC 420102007 and IEEE Std 14712000, 2011(March):1–46, 2011. doi: 10.1109/IEEESTD.2011.6129467.

[7] Microsoft Patterns & Practices Team. *Microsoft Application Architecture Guide*. Microsoft Press, 2009. ISBN 9780735627109.

[8] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, volume 123. 1996. ISBN 0131829572.

Contents

List of Figures

List of Tables

List of Code Fragments

List of Abbreviations

AD Architecture Description. 16

ADL Architecture Description Language. 6

AK Architectural Knowledge. 3

AP Architectural Perspective. 3, 38

AV Architectural View. 3

BPSL Balanced Pattern Specification Language. 7

CASE Computer-aided Software Engineering. 4

CLI Command-line Interface. 56, 59

CRC Class-Responsibility-Collaborator. 78

CRUD Create, Read, Update, Delete. 55, 61

DPIO Design Pattern Intent Ontology. 6

OWL Web Ontology Language. 6, 20, 22–24, 28, 52, 55

SPARQL SPARQL Protocol and RDF Query Language. 27, 28, 34, 35, 45, 47, 50–57, 61, 64, 67, 72, 88, 91, 94

Chapter 1

Introduction and related work

1.1 Software architectures and patterns

"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."

– Bass et al. [5]

Software architectures define structural elements in a system along with their interfaces and behaviour. The composition of the structural elements are specified in decision rules of the architecture.

Often, architectural elements are composed in ways that solve common particular problems [5]. To help capturing existing, well-proven experience in software development, patterns are used to build on the collective experience of skilled software engineers [8]. Patterns achieve reusability in software design. They originate from the architecture of buildings and have been ported to object-oriented software design by Gamma et al. [17] in 1995.

A distinction is made between design patterns and architectural patterns. Design patterns describe the detailed implementation of a component or a subsystem while architectural patterns describe the fundamental structure of a system. Architectural patterns define components, relationships between them and interaction mechanisms in order to provide a solution for a problem in a given context. Since architectural patterns require more trade-off decisions, they need modifications before applying it to a specific problem, and they become more complex to document in a formal and consistent way. For the concept "*pattern*", the definition from Buschmann et al. [7] is used, which extensively defines all components of *a pattern*, starting from the one-sentence characterisation of a pattern: *"A pattern is a solution to a problem that arises within a specific context"*.

1.2 Architecture documentation

Software architecture documentation is essential for efficiently sharing knowledge between developers and architects. It is important to make the retrieval as efficient as possible. In cases where developers use software development methodologies that don't focus on documentation, the importance of making the retrieval of information as fast as possible is even higher. The Agile Manifesto, for example, states:

> *"Comprehensive documentation is not necessarily bad, but the primary focus must remain on the final product-delivering working software. (...) So the distinction between agile and document-centric methodologies is not one of extensive documentation versus no documentation; rather a differing concept of the blend of documentation and conversation required to elicit understanding."*
>
> – Fowler and Highsmith [16]

Because of the focus shift from documentation to direct communication, good architectural design decisions rely on the architect's design experience and knowledge of the current system. Nord and Tomayko [30] illustrate this with an example and conclude that architecture-centric methods more efficiently use patterns and standards. Therefore, attention has to be paid to the quality of the documentation and the effectiveness and efficiency of its retrieval.

1.2.1 Common approach and its issues

The most common approach of documenting software architectures is the file-based approach [43]. In order to fit all the information of the architecture — also referred to as Architectural Knowledge (AK) — into files, a certain structure has to be respected in these files to keep the retrieval efficiency high. Otherwise, some AK might become redundant. One way to keep structure in the files, is to provide separate sections for separate subsystems, which can be done with Architectural Views (AVs). This is the recommended practice introduced by the IEEE-SA Standards Board [21]. Each view in the documentation addresses one or more of the concerns of stakeholders.

However, this only solves the problem of AK retrieval within one view, not the retrieval of knowledge scattered across different views. Rozanski and Woods introduce and discuss in [34], [36], and [35] the concept of Architectural Perspectives (APs). An AP helps the architect with practical guidance on how to achieve the desired quality properties of the system.

AVs give advise on how to describe a type of architectural structure, such as the functional structure, information structure, and so on. An AP on the other hand focusses on delivering a quality property such as performance, security and scalability. This means that an AP will have influence on multiple views. Using APs can thus be an effective way to organise knowledge such that the retrieval of AK scattered across different views becomes easier and more time-efficient.

Other issues of retrieving AK via a file-based approach are pointed out by de Graaf et al. [12]:

- Cross-referencing information is often limited or lacking completely, which increases the time needed to retrieve design or requirements details. If any referencing is present but certain AK is not present in the reference, this AK will not easily be found at all. Also, if some references are lacking in a certain requirement, changes to it might cause related requirements and designs to be inconsistent if the stakeholder cannot find them.
- If the overall structure is not consistent, some AK might become redundant and scattered across architectural views.
- If there is a large diversity of background and experience among the stakeholders reading the AK documents, one has to be careful with the used concepts and languages to avoid ambiguities.

Jansen et al. [23] discuss that, next to the AK retrieval issues, the process of updating AK may also experience difficulties. Documentation files have to be synchronised among a possibly large and distributed team in a formal way, such that, for example, versions shared by e-mail for a quick review are not being used by developers or other stakeholders.

Lastly, studies have shown that often engineers and architects are unclear about their responsibility in the documentation [42]. This can cause them to not update the documentation after their creation.

We can conclude that a more formal approach should be used. Most of the found literature in this domain addressed issues with documentation of the architecture of existing systems. However, theoretical documentation of practices that should be used in the design of an architecture can be slightly different, as will be described in Section 1.5.

1.2.2 Ontology approach

Many attempts have already been made to give an extra dimension in the documentation by using ontologies [31, 12, 1, 43, 3, 9]. An ontology is an explicit specification of a conceptualisation [18]. It defines a vocabulary for organising information such that the vocabulary and definitions are machine-interpretable. Software architecture documentation can benefit from ontologies by getting rid of the linearity that file-based approaches introduce. The most important example of non-linear information that ontologies can represent is relationships between decisions and requirements. Those relationships can be represented as references in the ontology, which makes it easier for stakeholders to find related information.

Also, because an ontology is defined to be machine-interpretable, automatic reasoning and querying can be implemented to help the architects or developers with questions like "*which components must be used to implement the non-functional requirement XYZ*". Osterwalder and Pigneur [32] use ontologies in a similar context in the field of business model engineering, where the use of ontologies is promoted for its compatibility with Computer-aided Software Engineering (CASE) tools. CASE tools reason about data, to aid manual research of the user. Moreover, it is compatible with the notation of formal semantics, so that a formal language can be extended with the given information [33].

1.3 Pattern documentation

Patterns are very useful in the field of software development, but one can only make good use of it when the retrieval of information about patterns can be done efficiently. In this section, several common used approaches are discussed.

1.3.1 File-based approach

As with architecture documentation, the approach of file-based pattern documentation will be able to document most of the information, but retrieval and maintenance of the information will become more difficult when the retriever does not have the appropriate knowledge to recognise the design problems for which the patterns are designed. Also, it is hard to efficiently reference to other patterns and index them in a file-based approach.

1.3.2 Formal languages approach

Using formal languages, one can conceptualise all components in a pattern documentation. Structures of components and relationships between them can be formulated in a theoretical sense. The biggest advantage of this is consistency. In classic file-based pattern documentations, it is common to find different structures of the components across literature. Also, it is common to find different vocabularies, which makes it hard to make links between concepts found in different literature sources.

The formal logic described by the formal language can mainly be divided in three categories:

1. Propositional logic, which is the less expressive, describes premises and logical connectives for these premises. Propositional logic is decidable.
2. Description logic, more expressive than propositional, usually decidable, consists of concepts, roles, individuals and their relationships.
3. Predicate logic (also known as first-order logic), is an extension of propositional logic that also allows quantification over the objects. This makes it more expressive but non-decidable.

The category of description logic languages forms the basis for ontology languages [20].

In a — relatively recent, compared with other literature found in this domain — study by Pahl et al. from 2009 (*Ontology-based Modelling of Architectural Styles*) [33], a model of architectural styles is designed based on the very basic foundations of description languages, in order to design a theoretical ontology for architectural styles. The aim of this publication is to develop an Architecture Description Language (ADL) with a description language as basis, so that the result can be integrated into existing ADLs. The approach in this work provides good insights on which concepts and relationships to consider in architectural patterns. However, this study only focusses on the theoretical part, whereas this master dissertation focusses on a practical (implemented) ontology, preferably written in Web Ontology Language (OWL) since this is nowadays one of the most common used ontology languages in the semantic web [24].

It is notable that both the concepts *architectural style* and *architectural pattern* are used in literature and that they are mostly used as synonyms. Avgeriou and Zdun [4] make a clear distinction between these concepts. First, architectural patterns are described as problem-solution pairs. Patterns also can contain more information about the context they are applicable in. The documentation and definition of the pattern is focused on the problem it solves. It is designed to work in pattern languages [7]. Next, architectural styles are described to be a category of components and components it consists of. Little attention is given to the problem the architectural style can solve. Its internal architectural configuration is more important since it can be used more easily as categorisation method. This master dissertation uses the concept *architectural pattern*, although *architectural styles* will also be considered as *patterns*. The ontology that will be developed, is designed to fit both definitions.

Kampffmeyer [25] is the first to propose a solution that uses an OWL ontology for indexing, classifying and formalising patterns. The focus of the ontology is on the problems that are solved by the considered design patterns, rather than the structure of the solution of the patterns (as is mostly done in the file-based approach). In other words, the formalisation will explain when to apply a design pattern — hence the subject of the paper "*Design Pattern Intent Ontology (DPIO)*". The author's purpose of formalising *when* to apply a design pattern, was to develop a wizard that will propose a design pattern if the user inputs a desired *intent*.

Unfortunately, the author only considers design patterns, which are less complex to formalise than architectural patterns. The approach of formalising concepts in this master dissertation will resemble the approach used in this work, but will be more general made so that it can contain more complex patterns.

Before Kampffmeyer [25], several attempts of Dietrich and Elgar [13] (2005), Mikkonen [28] (1998) and Maplesden et al. [26] (2002) were made to formalise design patterns in ontologies, but they only captured the structure of the design pattern, not its intent; so these ontologies have no use in the process of retrieving patterns for a given design problem. Henninger and Ashokkumar [19] designed a meta-model for an ontology that captures the structures of design patterns, and used it for automated axiomatic reasoning, but again lacks attention to the behaviour.
Taibi and Ngo [41] proposed a specification language in 2003 called Balanced Pattern Specification Language (BPSL) that aims to achieve equilibrium between both the structural and behavioural aspect of design patterns in order to have a formal specification of design patterns that is more complete than regular formalisations. Although this work deals only with design patterns, it was an inspiration for this master dissertation on architectural patterns because of the used relationships in the behavioural aspect of patterns.

1.4 Other approaches

Next to the file-based and formal languages approaches, some software visualisations exist that allow users to retrieve pattern information. Kampffmeyer [25] developed a wizard that can be used to visually construct a query on the underlying ontology, in order to retrieve pattern information. In each step in the wizard, the user can select which elements should be contained in the query, and which relationships those elements should have. Suter [40] uses ASP.NET MVC for the development of a web-based visualisation tool in which some information related to a pattern can be retrieved on a single page. The only supported information is currently a description of the pattern, along with a list of related patterns, books and internet links. The tool also allows to filter patterns on predefined categories, one of which is *Architectural patterns*.

These approaches with software visualisations are more accessible for users without knowledge of ontologies, while offering more dynamic functionality than file-based approaches. However, the usability and extensibility of every software visualisation tool is limited by its underlying data store. The more (and less complex) queries the data store supports, the more functionality the visualisation tool can provide to the end user.

1.5 Conclusion

As a first step in the literature research, several methods of documenting architectures were studied. The considered literature resources were mostly about already implemented architectures: they considered the architecture as a fixed input and tried to find the most optimal solution to document it. This research was a good introduction on the topic, but the found methods are not directly applicable to the problem statement at hand: finding an optimal way to document theoretical architectural patterns (patterns that yet have to be implemented).

After having studied the methods for documenting architectures, the scope was narrowed to the documentation of architectural patterns. Next, it was concluded that a formal approach — especially with ontologies — seems to be an ideal solution to document architectural patterns; just like it was with architecture documentation. Finally, some examples of visualisations were found that are of great value to users that want to query information of patterns. A combination of both (ontologies and a visualisation) will be discussed in the remaining chapters.

Chapter 2

Pattern classification

2.1 Problem statement and approach

As stated in Chapter 1, there is no consistent way in which architectural patterns are documented in literature. Different authors or publishers often use different properties of architectural patterns to categorise them. Authors then apply their categorisation techniques only to those patterns described in their work, which makes it often hard to categorise other patterns with their categorisation method.

Before making an ontology of architectural patterns, the categorisation methods were researched of some of the most relevant literature in this research domain, along with the most relevant patterns defined in their work:

- *Software Architecture in Practice* — Bass et al. [5]
- *Documenting software architectures: views and beyond* — Clements et al. [10]
- *Architectural Patterns Revisited - A Pattern Language* — Avgeriou and Zdun [4]
- *Microsoft Application Architecture Guide* — Microsoft Patterns & Practices Team [27]
- *Pattern Oriented Software Architecture* — Buschmann et al. [7]
- *Software Architecture: Perspectives on an Emerging Discipline* — Shaw and Garlan [37]

2.2 Generalisation

It was noticed that many patterns have different interpretations in different books. Also, two classification techniques based on a certain property of patterns are not necessarily compatible with each other. For example, Avgeriou and Zdun [4] and Clements et al. [10] both use *Viewtypes* as pattern property for classification and use different categories. Table 2.1 lists the categories of their categorisation technique.

Clements et al. [10]	**Avgeriou and Zdun [4]**
Module	Distribution
Allocation	Adaptation
→Work assignment	Data-centered
→Deployment	User-interaction
Component-Connector	Data-flow
	Layered
	Component-Interaction
	Language Extension

Table 2.1: List of classification categories (viewtypes) for [10] and [4].

Note that these viewtypes are not compatible. We can clearly see that the viewtypes in the left column are of a higher level than those in the right column. However, the viewtype *Component-Connector* defined in Clements et al. [10] is very similar to the viewtype *Component-Interaction* defined in Avgeriou and Zdun [4]. There are a number of patterns that belong to both categories, although the definition in the literature of both categories (viewtypes) is slightly different. For this reason it is important, when trying to make a generalised categorisation technique, to make sure that every relationship between a category and a pattern, and every category itself, is well annotated with the proper reference to the author of the relationship or category. Otherwise one could confuse categories and make false categorisation decisions.

Although pattern definitions differ from one book to another, which is the reason to annotate each pattern with their respective author in an ontology or software tool, one wishes to treat them all the same when browsing through relevant patterns for a software design problem. Therefore the categorisation schema was made as follows. The end result of the classification schema is included in Appendix B on page 102. This schema can also be found on GitHub[1] or obtained at e-mail request[2].

- In the categorisation schema, all patterns are indicated with boxes, regardless of which author defined the pattern.
- All categories, from all authors listed above, are indicated with boxes with dotted lines. Each black box is a root node for boxes with dotted lines — they represent the categorisation method of one book/author.
- Each relationship is annotated with the name of the author that defined that relationship. A relationship can be:
  - A definition of a category: for example the Microsoft Patterns & Practices Team [27] takes as categorisation method *"Area of focus"* and defines four categories. The relationship between the categorisation type *"Area of focus"* and the categories are annotated with the author reference. In order to be visually clear, annotations on author references are made with colour indications.
  - An assignment of a pattern to a category.
  - A relationship between two patterns, see Section 3.2.2.

  The type of relationship is indicated with an arrow line style. For example, the relationship *"pattern X belongs to category Y"* is indicated with a straight line, while a dotted line is used for the relationship *"pattern X is used in pattern Y"*. Figure 2.2 defines these types.
- When two patterns or categories are defined by different authors in exactly the same way but they use different terminology, the patterns are connected with an *"is equal to"* relationship, while separation of relationships from and to those patterns is preserved.

Figure 2.1 illustrates a simplified version of the schema, where only the main categories are used (no subcategories or patterns). The black boxes indicate a categorisation type, one for each author. The text in the black boxes reveals how

[1]See https://github.com/wpinnoo/thesis/blob/master/schema.pdf
[2]Wouter Pinnoo, pinnoo.wouter@gmail.com

Author	**Categorisation type**	**Concept that is considered to be relevant for this categorisation**
Microsoft Patterns & Practices Team [27]	Area of focus	Purposes
Clements et al. [10]	Viewtypes	Viewtypes
Bass et al. [5]	Viewtypes	Viewtypes
Shaw and Garlan [37]	Style category	Structures
Buschmann et al. [7]	Problem category	Purposes
Avgeriou and Zdun [4]	Viewtypes	Viewtypes

Table 2.2: Mapping of categorisation types to ontology concepts

this type is described. In order to combine or generalise the types as much as possible (without losing any information), these types are mapped to concepts that will be used in the ontology, as will be described in Chapter 3. Table 2.2 lists these mappings.

Figure 2.1 also depicts the annotation of authors to the relationships *"category belongs to categorisation type"*. Some relationships are indicated with more than one colour, as is the case with the *Module*, *Allocation* and *Component-Connector* viewtype. The more frequent a categorisation type is used by different authors, the more likely this type is an important one for software architects to use when browsing through patterns.

Next to black boxes, there are other types of boxes. Figure 2.2 lists a legend of these types. Specific categories that inherit from a classification method are indicated with dotted lines. Patterns that only serve as patterns and not as (sub)category are indicated with straight lines. In some cases, what one literature resource describes as a *pattern* is described in other resources as *a category of patterns*. For example, the *Broker* pattern is in most literature sources described as a standalone pattern, but Buschmann et al. [7] treat *Broker* as a category of variants of the *Broker* pattern, including the *Trader system* pattern, the *Adapter broker system* pattern and *Callback broker system* pattern. Such patterns are indicated with dot-line-dot-lines, as seen in Figure 2.3.

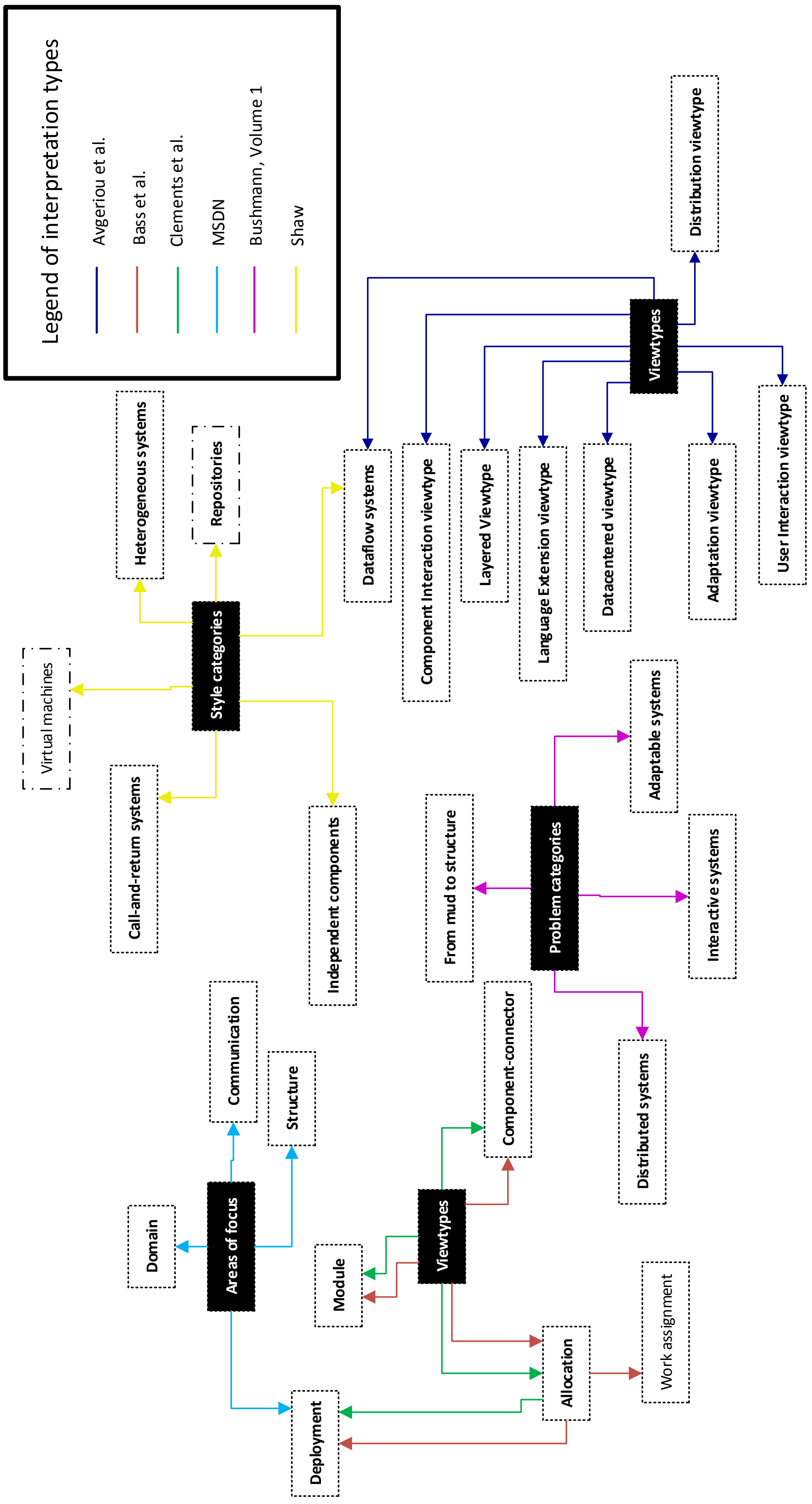


Figure 2.1: Main categories in the classification schema

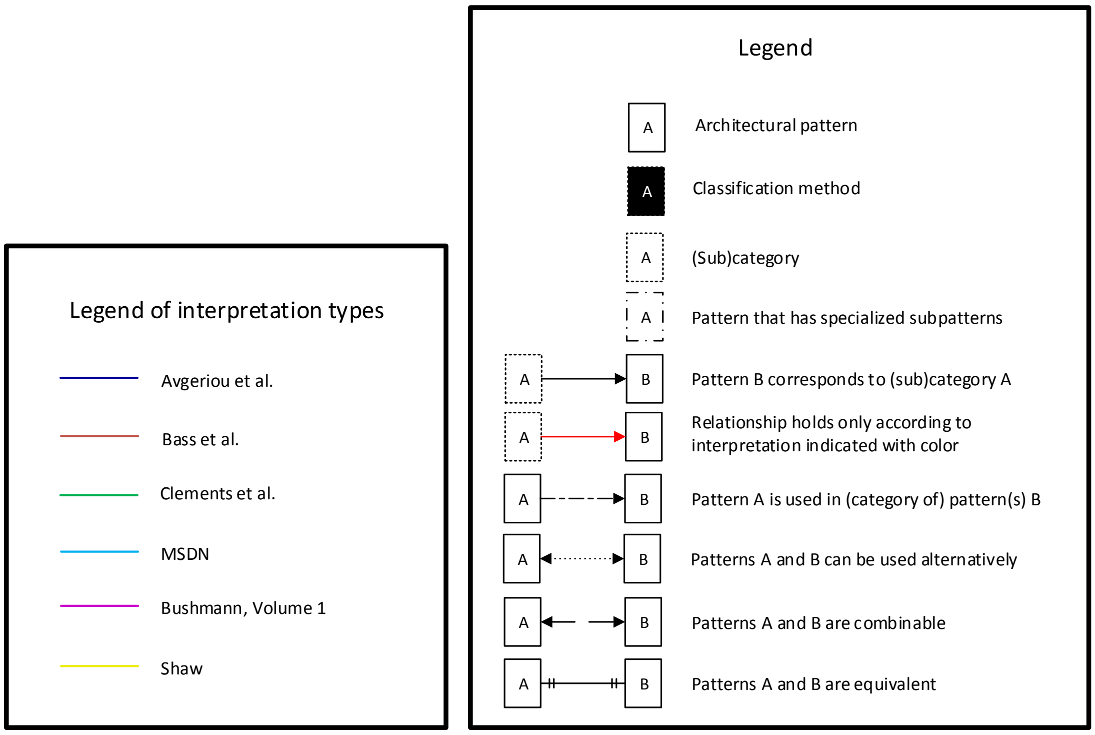

Legend of interpretation types
Avgeriou et al.
Bass et al.
Clements et al.
MSDN
Bushmann, Volume 1
Shaw
Legend
Architectural pattern
Classification method
(Sub)category
Pattern that has specialized subpatterns
Pattern B corresponds to (sub)category A
Relationship holds only according to interpretation indicated with color
Pattern A is used in (category of) pattern(s) B
Patterns A and B can be used alternatively
Patterns A and B are combinable
Patterns A and B are equivalent

Figure 2.2: Legend of the classification schema

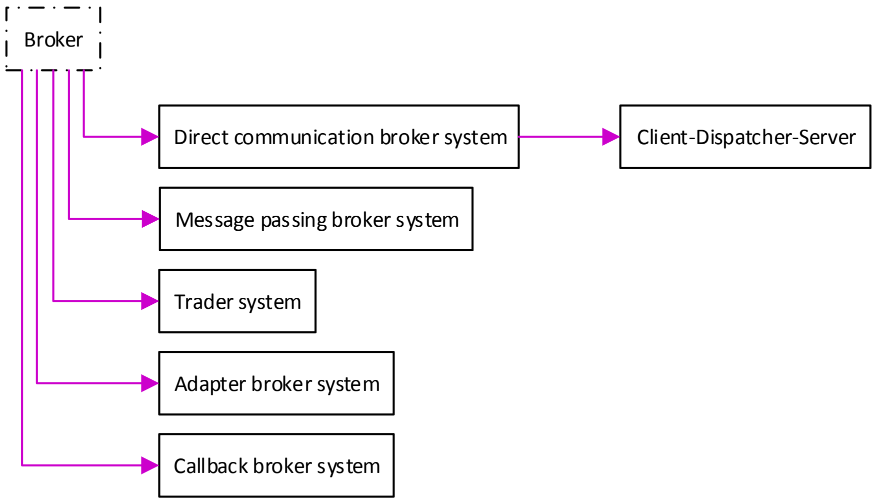


Figure 2.3: Illustration of the Broker pattern as both pattern and category

2.3 Conclusion

In this chapter, a classification of architectural patterns was made, based on information found in some of the most used literature sources in this domain. Because those patterns were documented in different ways, a generalisation was made for the structure that categorises the patterns. A classification schema was presented that captures all of the gathered information — from pattern relationships to resource links.

With the insights from the gathered information, an ontology will be designed in Chapter 3. As discussed in Chapter 1, ontologies define a vocabulary for organising information such that the result is machine-interpretable. This is the ideal approach for storing the gathered information since tools can be built that apply automatic reasoning on the information to present inferred information to architects. In addition, ontologies are the common way to represent information in the semantic web, which implies that the designed ontology can be integrated with other ontologies in the semantic web. Conversely, new ontologies can be built that extend the existing ontology.

Another approach to store the gathered information, as an alternative to ontologies, would be representing the information in a traditional object-oriented database. The design of the database would then be very similar to the design of this ontology — the same classes can be constructed. However, a traditional database is not able to apply automatic reasoning nor is it compatible with information found in existing ontologies or information in the semantic web, which makes it less future-proof.

Chapter 3

Ontology design

In this chapter, the design choices of the ontology are explained, along with a descriptive illustration of the ontology. The code of the ontology can be found on GitHub[1] or obtained at e-mail request[2].

3.1 Concepts and components

As a starting point for the components to be used, both concepts from the ISO standard and insights from the pattern classification from Chapter 2 were used.

The ISO/IEC/IEEE 42010 standard from 2011, *Systems and software engineering – Architecture description* [22] defines its scope as *the matter in which architecture descriptions of systems are organised and expressed*. It defines the concepts that can be used to define Architecture Descriptions (ADs). An AD is a collection of definitions and schemas that describe a concrete software architecture. Figure 3.1 illustrates the concepts defined in the ISO-42010 standard. An approach with these concepts is also used by Rozanski and Woods [36].

[1] See https://github.com/wpinnoo/thesis/blob/master/patterns.owl
[2] Wouter Pinnoo, pinnoo.wouter@gmail.com

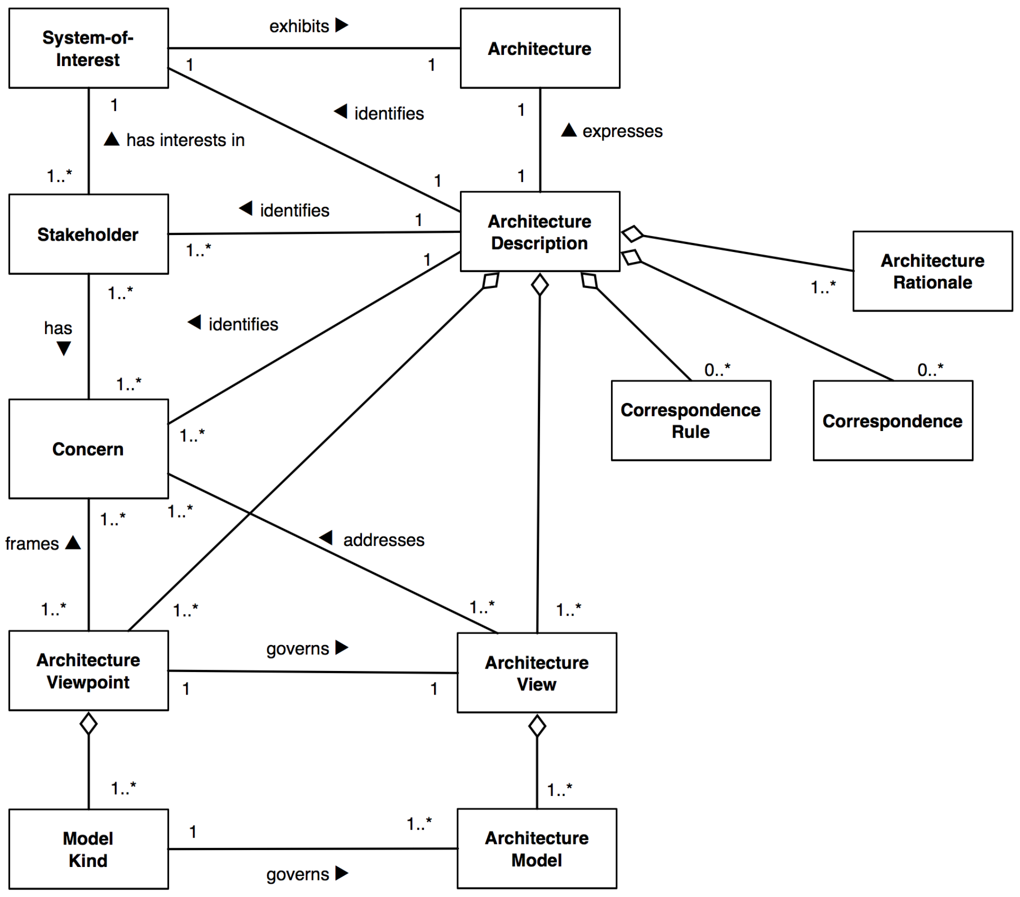


Figure 3.1: Concepts of the ISO 42010 standard. Source: [22]

3.1.1 Purposes

We can identify interesting concepts to use in the classification. For example, in Table 2.2 (page 12) a mapping was made between categories defined in literature and more general category names. Amongst others, *Purposes* were used. Purposes can be seen as type of a concern of a stakeholder. This fits the definition in the ISO standard since they describe a concern as a part of an architecture description. Another type of concern that will be defined in the ontology is a *Quality Attribute* (see Section 3.1.3). The concept *architecture description* will be used for an architectural pattern documentation.

3.1.2 Views, viewtypes and viewpoints

Table 2.2 (page 12) also uses *Viewtypes* as categories. Viewtypes are categories of Views [4], which are defined in the ISO standard. This implies that an architectural pattern doesn't have a direct connection with a viewtype — it only relates to a viewtype through his specific view. Since some books define variants of patterns where the pattern variant is actually just an extension of the view of the other pattern, the ontology will provide inheritance of views. The same holds for purposes and structures.

The concept *Viewpoint* is defined in the ISO standard as the component that *establishes the conventions for constructing, interpreting and analyzing the view to address concerns framed by that viewpoint*. This concept is also used in the ontology. Although some literature use a fixed list of viewpoints to categorise the views of patterns in, the ISO standard states that no particular viewpoints should be used. Therefore the ontology does not impose a restriction on which viewpoints should be used.

Avgeriou and Zdun [4] also used views to categorise patterns on. The exact categorisation however is implemented in a different way: the authors consider a pattern as a specialisation of a viewpoint because a viewpoint describes the types of the elements and relationships to be used in the views; and views contain the elements and relationships to be used in a pattern. In this ontology however, viewpoints are just a property of a pattern. In case there is a hierarchy in viewpoints, this can be implemented with an inheritance relationship. The same holds for views. As the authors describe, views can either be considered as a high-level concept to which multiple patterns can be assigned; or as a component with a one-to-one mapping with patterns. The latter requires again that inheritance of views can be implemented. This approach is chosen because it allows to include view-specific information in the ontology by having a view component for each pattern.

3.1.3 Other components

Some concepts of the ISO standard were found irrelevant in the categorisation methods of the considered literature sources, for example *Correspondence Rule*, *Correspondence* and *Architecture Rationale*; while other concepts were added that were not contained in the ISO standard. Those concepts are:

- *Example*: allows to add specific examples to each pattern object in the ontology. The content of the example can be added as String content or as URIs to the *Example* object.
- *Structure, Collaborator, Responsibility*: allows to define the internal structure of the pattern. A structure is defined by its collaborators and their responsibilities. Descriptions of these components can be added as String annotations on the objects.
- *Tactic, Quality Attribute*: these two components are some of the most important ones for the architects. Architects will mainly use this ontology (and its web tool, see Chapter 5) for retrieving patterns given a quality attribute they want to achieve. Quality attributes are achievable with tactics.

Figure 3.2 illustrates the final concepts that are used in the ontology. Components with a grey background are those that will be used as categorisation components. Figure 3.3 explains the used arrow types. The first, filled arrow defines a relationship. In some cases there is an inverse relationship shown in the diagram, such as *pattern* → *comprises* → *tactic* for the relationship *tactic* → *augments* → *pattern*. In cases where there is no explicit inverse relationship defined, an inverse relationship with a straightforward name will be added to the ontology, e.g. *isAchievedBy* for the relationship *achieves*. Having inverse relationships aids querying the ontology. An arrow that is not filled defines an specialisation. For example, a "purpose" *is* a type of a "concern", and so is a "quality attribute". A diamond arrow indicates a *has* relationship: a pattern has some examples, has a structure, etc.

The pattern class has some relationships to itself. These relationships are used to define alternative patterns, patterns that use other patterns, etc. Some other components have the *isChildOf* and *isParentOf* relationship. These are used for defining variants. This will be further explained in Section 3.2.1.

Figure 3.2 is the basis for the actual implementation of the ontology and is a great utility in the process of designing queries. For example, we can follow all arrows to find out which relationships will have to be queried to retrieve a pattern if a certain viewtype is given.

3.2 Ontology implementation

The OWL is chosen for the implementation of the ontology because it is the most widely used language in the semantic web [44]. The ontology was developed with the use of the Protégé tool [39]. This tool allows to easily edit and construct components, relationships and annotations in the ontology. It supports a live reasoner and visualises all inferred data (reasoning and inferred data are discussed in Section 4.4). Although Protégé handles the generation of the OWL code based on the user input in the tool, the OWL code will be discussed in this section since future developers of this ontology might work on the raw ontology file or use other tools.

3.2.1 Components

The components depicted in Figure 3.2 are added as classes in the ontology. The expression that *Purposes* and *Quality Attributes* are *Concerns* is implemented with these components being subclasses. Every specific implementation is an ontology instance, also called an individual. This makes it impossible to have inheritance of such objects. For example, a variant of pattern X can not be implemented as a sub-instance of the instance of pattern Y. Yet this approach was chosen because it was easier to define relationships between objects. The problem of inheritance of patterns (and also other concepts, such as purposes, viewtypes, tactics and structures) is solved by using explicit object properties *isChildOf* and *isParentOf*. These relationships are defined as transitive and inverse of each other, which makes it a usable alternative

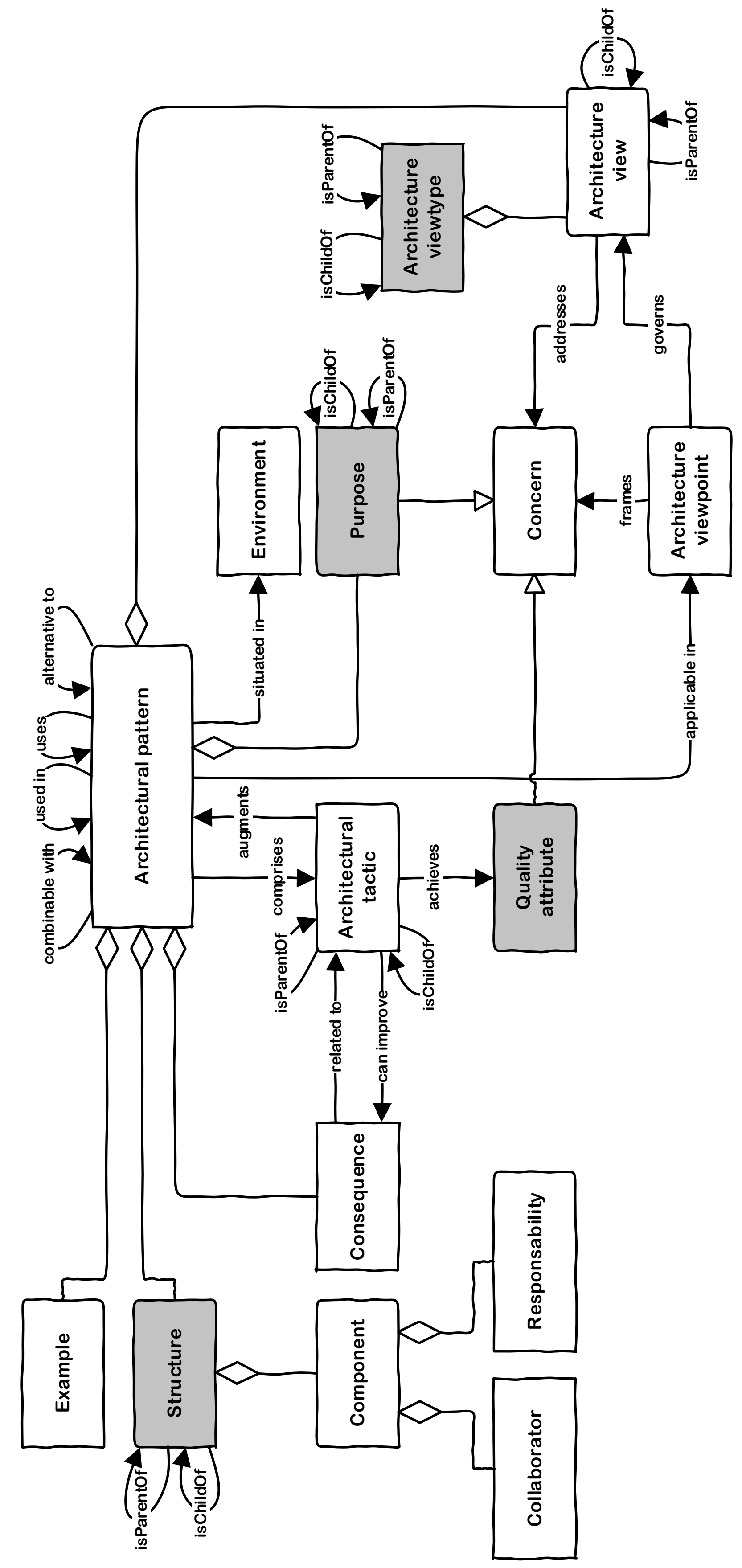


Figure 3.2: Schema of the ontology components. Grey boxes are components used for categorisation. A legend for this figure is depicted in Figure 3.3.

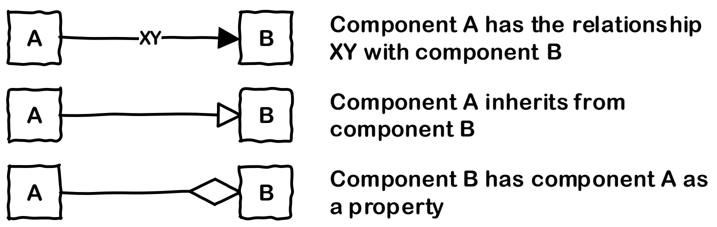

A
XY
B
Component A has the relationship XY with component B
A
B
Component A inherits from component B
A
B
Component B has component A as a property

Figure 3.3: Legend of the ontology schemas

to class inheritance. Listing 3.1 shows the implementation of an individual in OWL syntax. The type of the individual is defined with the `rdf:type` property. The `label` property is defined to provide readable names in the ontology that can be used in visualisations of the ontology.

```
root@web:~# sed -n '3075,3079p' /root/patterns.owl
    <ns1:NamedIndividual rdf:about="&ns3;BrokerImplStructure">
        <rdf:type rdf:resource="&ns3;Structure"/>
        <ns2:label>Broker</ns2:label>
        <belongsTo rdf:resource="&ns3;BrokerPattern"/>
    </ns1:NamedIndividual>
```

Listing 3.1: OWL (.owl) syntax

The names of instances are chosen to use suffixes that indicate the type of the instance. For example, the `BrokerPattern` is an instance of the class `Pattern`. For each pattern instance, a structure, purpose, view and viewpoint instance are created irrespective of whether they have useful content. This simplifies further usage of contents of the ontology: when defining relationships, one can assume that the respective instances already exist. The implementations are suffixed with `...ImplStructure`, `...ImplPurpose`, etc. This way we can distinguish instances that belong to patterns from instances only serving as categorisation type.

As an example, consider again the Broker pattern as depicted in Figure 3.4. It has a variant called *Callback Broker System*. Its purpose, `CallbackBrokerSystem-ImplPurpose` is a child of the `BrokerImplPurpose`, since according to Table 2.2 the categorisation type ("problem category") of this author is mapped to purposes. The `BrokerImplPurpose` is a child of `DistributedSystemsPurpose`. We recognise by the name of this last instance that this is a categorisation type that doesn't belong to any pattern and is thus a root category.

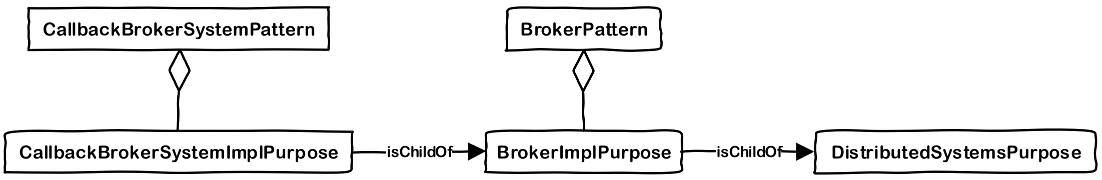
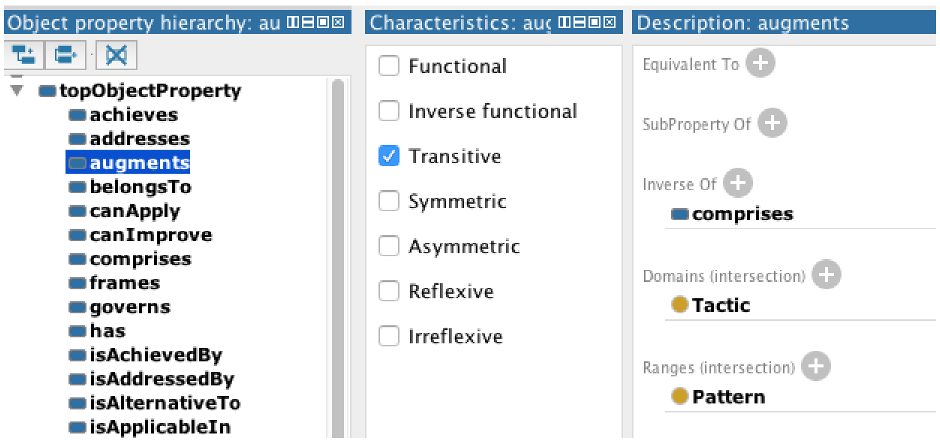

CallbackBrokerSystemPattern
BrokerPattern
CallbackBrokerSystemImplPurpose
isChildOf
BrokerImplPurpose
isChildOf
DistributedSystemsPurpose

Figure 3.4: An example of the design of the ontology that shows variants of the Broker pattern.


Object property hierarchy: au

- topObjectProperty
  - achieves
  - addresses
  - augments
  - belongsTo
  - canApply
  - canImprove
  - comprises
  - frames
  - governs
  - has
  - isAchievedBy
  - isAddressedBy
  - isAlternativeTo
  - isApplicableIn

Characteristics: aug

- [ ] Functional
- [ ] Inverse functional
- [✓] Transitive
- [ ] Symmetric
- [ ] Asymmetric
- [ ] Reflexive
- [ ] Irreflexive

Description: augments

Equivalent To

SubProperty Of

Inverse Of
comprises

Domains (intersection)
Tactic

Ranges (intersection)
Pattern

Figure 3.5: Defining properties in Protégé

3.2.2 Relationships (properties)

The arrows on Figure 3.2 are implemented using object properties. These were defined using Protégé (see Figure 3.5) and result in the OWL code of Listing 3.2. Some object properties are marked with the transitive property, and an inverse relationship is defined (if applicable). This is necessary for reasoners to work correctly (see more on reasoning in Section 4.4). Domains and ranges are defined in objects properties as a constraint, to maintain correctness of the ontology, in case it is modified.

```
root@web:~# sed -n '116,121p' /root/patterns.owl
    <ns1:ObjectProperty rdf:about="&ns3;augments">
        <rdf:type rdf:resource="&ns1;TransitiveProperty"/>
        <ns2:range rdf:resource="&ns3;Pattern"/>
        <ns2:domain rdf:resource="&ns3;Tactic"/>
        <ns1:inverseOf rdf:resource="&ns3;comprises"/>
    </ns1:ObjectProperty>
```

Listing 3.2: Defining properties in OWL

3.3 Resource annotations

Annotations are used for adding metadata to instances, classes or object properties. Such metadata can be plain-text, URIs, or links to DBpedia resources. DBpedia is a publicly available ontology that is a representation of Wikipedia content in the semantic web [6]. DBpedia ontology URLs can be included in this patterns ontology so that the Wikipedia abstract of a pattern can be used to describe the pattern.

Figure 3.6 illustrates how resources are linked to items in the ontology. The annotation is implemented with OWL Axioms. Axioms define the triple

annotated source

→ *does something with the annotated property*

→ *to the annotated target*

in which the source and target can be anything — mostly a class or an individual. The annotated property is mostly an object property (a relationship). By annotating object properties, we can attach links to external resources directly to the relationship between objects. It is also possible to attach an axiom with resource links to the `rdf:type` property, which is the property that defines to which class an individual belongs. This is used for attaching information to an object itself: e.g. "*which resource defines that modifiability is a quality attribute?*".

The actual linking of resource information to an annotation property is done by adding a *Resource* individual and by defining the relationship *axiom* → *resourceAuthor* → *resource-individual*. This resource individual will contain all metadata. It contains data properties that can consist of an ISBN number, plain-text, an URI or a DBpedia URI. The latter is an ontology URI that refers to a Wikipedia page. By including this ontology URI in the pattern ontology, one can easily retrieve e.g. the Wikipedia abstract and use it as description of the retrieved object.

3.4 Conclusion

In this chapter, research was done on which components should be contained in the ontology and on which way data can be included in those components. A selection

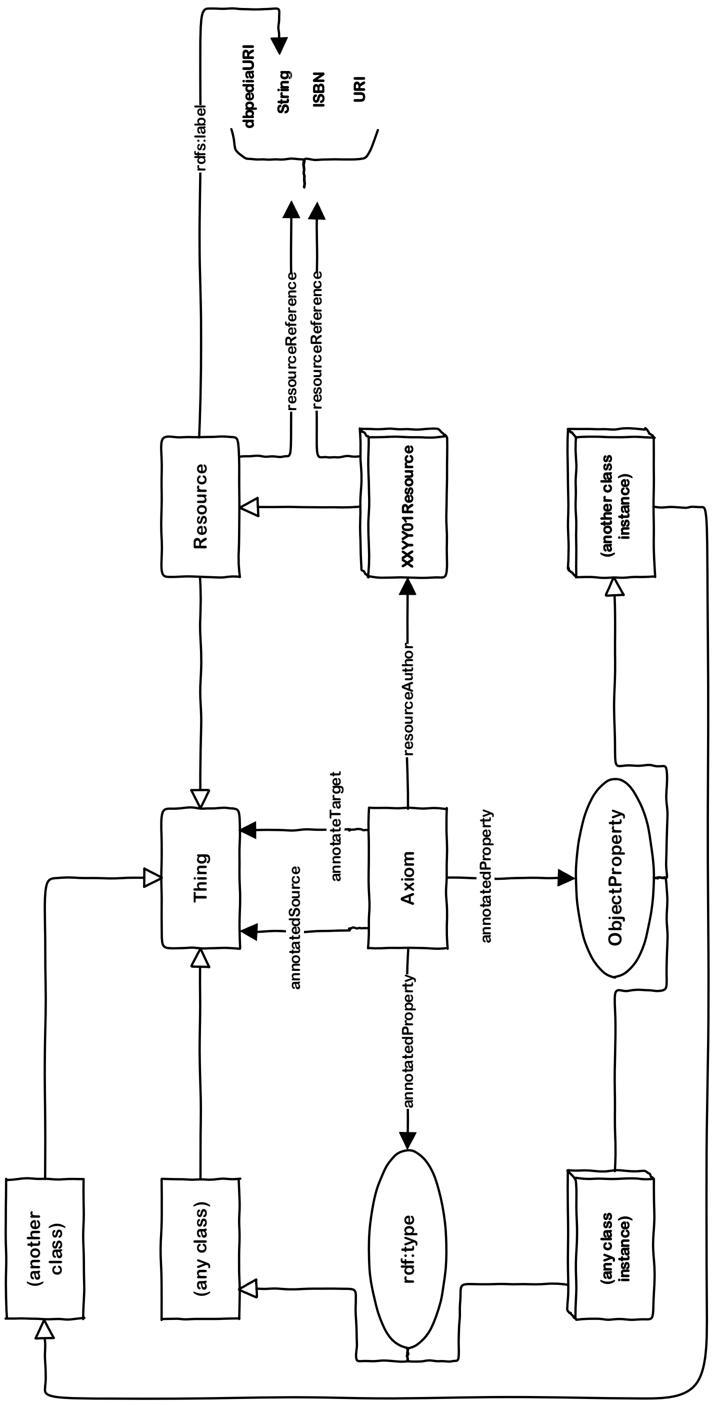


Figure 3.6: Resource linking in the ontology

was made based on components found in literature, and a description was given on the proposed approach to include information in the ontology about both the actual content and the related resources.

Chapter 4

Ontology querying

4.1 Introduction

A description of the designed ontology was given in the previous chapter. This chapter will focus on how to query the ontology. Constructing queries on the ontology is an important part of this master dissertation as it, for the first time, reveals both the possibilities of the ontology (which information can be retrieved and which input information is needed) and its flexibility (which modifications are needed if the ontology is extended).

4.2 Querying language

4.2.1 SPARQL

The SPARQL Protocol and RDF Query Language (SPARQL) [46] was chosen as querying language for the ontology as it is the most common used one in the semantic web compared to other querying languages like SWRL [45] and SQWRL [38]. SPARQL is developed and maintained by the World Wide Web Consortium [46]. A SPARQL query can be executed on a SPARQL engine that operates on either a database or an ontology file. Both options are researched in detail in Section 6.1 (page 52). Only a simple SPARQL engine operating directly on an ontology file is considered in this chapter.

4.2.2 Query syntax

A SPARQL query is similar to SQL but uses triples in the *SELECT-WHERE* clauses. Listing 4.1 shows the syntax of a query used in this master dissertation.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://thesis.wouterpinnoo.be/patterns.owl#>
SELECT ?subject ?object WHERE {
    ?subject :has ?object .
}
ORDER BY ?subject
```

Listing 4.1: Syntax of a SPARQL query

All prefixes are defined in the first part. Some ontology relationships are defined in the OWL ontology URL, so these URLs are defined as prefixes. For example, the relationship *individual X is an instance of class Y* is defined with the object property `rdf:type`. An individual is of the type `owl:NamedIndividual`. These prefixes will be used later on, in the construction of queries.

Next, a *SELECT-WHERE* clause is defined. After the `SELECT` statement, a list separated with spaces is given of all objects that will be included in the query response. In the `WHERE` clause a list of statements is given separated with a dot. Each statement is of the form

?subject :relationship ?object .

and thus always describe the expression *a subject has a relationship with an object*. The question mark `?` before `subject` and `object` indicate that these are variables. Instead of variables, one might want to use the expression *any* in a statement, rather than strictly defining a variable. This can be done with brackets, as seen in Listing 4.2 on line 8.

A variable can be used in multiple statements to specify constraints on the variable. For example, Listing 4.2 constraints that the variable `subject` is a pattern, is applicable in at least one viewpoint, and has the relationship *comprises* with the variable `object` (that is a tactic). The relationship *comprises* starts with a colon (`:`), which indicates that the prefix "`:`" referring to the URL http://thesis.wouterpinnoo.be/patterns.owl# must be used for this relationship. In other words, this relationship is defined in the ontology of this master dissertation. The same holds for the colon before Pattern, Viewpoint, Tactic and isApplicableIn.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://thesis.wouterpinnoo.be/patterns.owl#>
SELECT ?subject ?object WHERE {
    ?subject a :Pattern .                       # <a> is short synonym for <rdf:type>
    ?subject :isApplicableIn [a :Viewpoint] .
    ?subject :comprises ?object .
    ?object a :Tactic .
}
ORDER BY ?subject
```

Listing 4.2: Constraints in a SPARQL query

A specific class instance can be bound to a variable. This is useful for retrieving information that is related to a certain class instance. An example for the BrokerPattern instance is given in Listing 4.3.

```
(...)
 BIND (<http://thesis.wouterpinnoo.be/patterns.owl#BrokerPattern> AS ?subject .
(... do something with ?subject ...)
```

Listing 4.3: Binding an object to a variable

In Section 6.1.1 (page 52), an example and explanation of the output of such queries are given.

4.3 Examples

4.3.1 Categories

As a first example, all quality attributes are retrieved from the ontology along with those tactics that achieve any quality attribute defined in the ontology. The example is given in Listing 4.4. When querying with the relationship "*Quality attribute* is achieved by *Tactic*" on line 10, a union is taken with the inverse relationship *achieves*.

This is to avoid missing some entries in case the query is executed without reasoner. More on this will be discussed in Section 4.4. No duplicate results will be generated since the `DISTINCT` keyword was provided in the `SELECT` statement.

Line 11 in this example illustrates how readable names for objects are retrieved. Since labels are data properties (String objects), the label can be retrieved with a direct relationship between the object and the label.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX owl: <http://www.w3.org/2002/07/owl#>
3  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
5  PREFIX : <http://thesis.wouterpinnoo.be/patterns.owl#>
6  SELECT DISTINCT ?qa ?tactic ?qalabel ?tacticlabel
7  WHERE {
8      ?qa a :QualityAttribute.
9      ?tactic a :Tactic.
10     {?qa :isAchievedBy ?tactic} UNION {?tactic :achieves ?qa}.
11     ?qa rdfs:label ?qalabel .
12     ?tactic rdfs:label ?tacticlabel .
13 }
14 ORDER BY ?qa ?tactic
```

Listing 4.4: Example of querying quality attributes and tactics

The above example is the least complex one for retrieving categories because it is already known what the root node is: quality attributes. Quality attributes don't have internal child-parent relationships and quality attributes are not directly linked to patterns. Querying category details of structures and purposes however requires explicit retrieval of root nodes. Structures for example are directly linked to patterns with the "`has`" relationship, and structures can inherit from each other. The query must therefore retrieve the root nodes and those structures that are *not* linked to a pattern. This automatically implies that those structures have no parent structure. An example query is listed in Listing 4.5.

This example illustrates also the use of nested `SELECT` statements. These are in this case used to make a union of two `SELECT` statements. The first one queries the ontology for all combinations of *(structure, substructure)* in order to gain all knowledge of the structure inheritance. An asterisk ($\star$) is added to the relationship to indicate transitivity. To aid further processing of the result of this query, the second `SELECT` statement queries the root nodes: all structures that don't have any related pattern. Those entries in the response of this query will be recognised by the value "0" for the variable ?structure.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://thesis.wouterpinnoo.be/patterns.owl#>
SELECT DISTINCT * {
    {SELECT *
    WHERE {
        ?structure a :Structure.
        ?substructure a :Structure.
        ?substructure :isChildOf* ?structure.
        FILTER (?substructure != ?structure).
        ?structure rdfs:label ?structurelabel .
        ?substructure rdfs:label ?substructurelabel .
    }}
    UNION
    {SELECT ("0" as ?structure) ("0" as ?structurelabel)
     ?substructure ?substructurelabel
    WHERE {
        ?substructure a :Structure.
        FILTER NOT EXISTS {
            [a :Pattern] :has ?substructure
        }.
        FILTER NOT EXISTS {
            ?substructure :belongsTo [a :Pattern]
        }.
        ?substructure rdfs:label ?substructurelabel .
    }} .
} ORDER BY ?structure ?substructure
```

Listing 4.5: Example of querying quality attributes and tactics

4.3.2 Resources

As discussed in Section 3.3, links to external resources are included in the ontology. An example of querying such resource links is given in this section.

Links to resources are implemented using axioms. Axioms are annotations on classes, individuals or relationships (properties). As depicted in Figure 4.1, axioms have build-in properties owl:annotatedSource, owl:annotatedTarget and owl:annotatedProperty. Listing 4.6 shows the ontology code for such an axiom. In the case of linking resources to ontology relationships (properties), the source and target properties refer to the ontology class instances that are taking

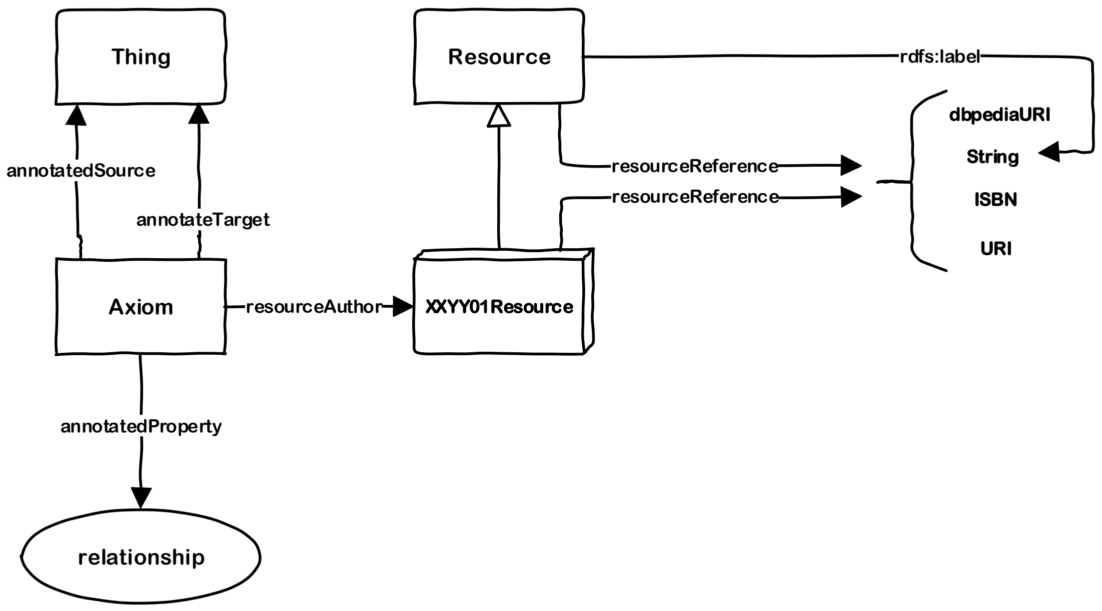


Figure 4.1: Resource axioms (simplified)

part in the relationship. The `annotatedProperty` property will directly refer to the ontology property that is defined between the source and the target, e.g. `:augments`. The `:resourceAuthor` property is used to link an axiom to a resource object. It is the resource object's responsibility to contain all resource links. The resource object is an instantiation of a Resource class that can contain more general information that is valid for all its resource objects. For example, a subclass of the Resource class can be dedicated to a certain book or author and can thus contain the ISBN number of the book along with a readable name for that book (`rdfs:label`). Resource objects belonging to that class will define more specific information, only valid for the axiom the object is involved in. For example, a resource object can contain a `String` reference that contains the page number of the book on which the information required by the axiom is found.

```
root@web:~# sed -n '530,535p' /root/patterns.owl
    <ns1:Axiom>
        <ns1:annotatedSource rdf:resource="&ns3;AbstractCommonServicesTactic"/>
        <resourceAuthor rdf:resource="&ns3;Bass0Resource"/>
        <ns1:annotatedTarget rdf:resource="&ns3;LayeredArchitecturePattern"/>
        <ns1:annotatedProperty rdf:resource="&ns3;augments"/>
    </ns1:Axiom>
```

Listing 4.6: Axiom syntax example

The querying of resource links can thus be categorised in two types:

1. queries on axioms of specific relationships between two ontology individuals;
2. queries on more general information: resources defined by the Resource classes.

These types are discussed in respectively Section 4.3.2.1 and 4.3.2.2.

4.3.2.1 Resources linked to specific relationships

When querying for relationships between two objects, the resources can be retrieved by looking for relevant axioms. Listing 4.7 is an example where the annotations of the relationship *IncreaseSemanticCoherenceTactic augments a Pattern* are queried. The relevant axioms are queried using the `annotatedSource` and `annotatedSource` properties. This axiom refers to the relevant *Resource* instantiation with the property `:resourceAuthor`. All resource information is linked to this *Resource* instantiation with `resourceReference` properties. In this case, the property links to a `String` property, which probably contains a page number or text reference in a book. The reference of the book itself is discussed in Section 4.3.2.2.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://thesis.wouterpinnoo.be/patterns.owl#>
SELECT ?item ?itemType ?label ?object ?resourceLink ?resourceLabel ?resourceISBN
    ?resourceComment
WHERE {
    BIND
        (<http://thesis.wouterpinnoo.be/patterns.owl#IncreaseSemanticCoherenceTactic>
        AS ?object).
    ?item a :Pattern.
    ?object :augments+ ?item .
    ?axiom :resourceAuthor ?resource.
    ?axiom owl:annotatedSource ?object.
    ?axiom owl:annotatedTarget ?item.
    ?axiom owl:annotatedProperty :augments.
    ?resource a ?resourceType.
    OPTIONAL {
        ?resource :resourceReference ?resourceComment.
        FILTER ( datatype(?resourceComment) = :String).
    }
}
```

Listing 4.7: Example of querying resource links on object properties

4.3.2.2 Resources linked to *Resource* classes

A subclass of the Resource class can have a readable name defined with the property `rdfs:label`. This label can be retrieved with a simple property query as seen in the example of Listing 4.4 on line 11.

Other resource links can be retrieved in a similar way. After the resource individual is found, its corresponding class is retrieved as seen in Listing 4.8 on line 3. The `resourceReference` property is used to link those classes to data properties, such as an ISBN number or a DBpedia ontology URL.

```
1  SELECT ?resource ?type ?resourceLink ?resourceLabel ?resourceISBN
2  WHERE {
3      ?resource a ?type .
4      ?type rdfs:subClassOf* :Resource .
5      ?type :resourceReference ?resourceISBN.
6      FILTER ( datatype(?resourceISBN) = :ISBN).
7  }
```

Listing 4.8: Example of querying resource links on Resource classes

If the referenced resource link has the datatype `:dbpediaURI`, this URI can be used to retrieve more information. This URI is a link to an object from the DBpedia ontology. This ontology can contain abstracts of Wikipedia pages. Listing 4.9 illustrates how Wikipedia abstracts are retrieved when in the previous query a DBpedia URI was retrieved. This query is executed on the DBpedia SPARQL engine.

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
SELECT ?abstract ?wikipediaUrl
WHERE {
    BIND (<’ + dbpediaUrl + ’> AS ?object) .
    ?object dbpedia-owl:abstract ?abstract .
    FILTER(langMatches(lang(?abstract),"en")) .
    ?object foaf:isPrimaryTopicOf ?wikipediaUrl .
}
```

Listing 4.9: Retrieving a Wikipedia abstract via DBpedia

4.4 Reasoning

A SPARQL engine executes a query on an ontology. Depending on the configuration of the engine, the ontology can be either static (an ontology file) or a database containing all information provided in the ontology file.

Consider the expression

Variable X has relationship Y with variable Z

which can easily be converted into a SPARQL query as seen in Section 4.2.2. When executing this query, the engine will by default look for explicit occurrences of this expression. Although the ontology might contain a relationship Y' that has been marked in the ontology as being the inverse relationship of Y, expressions with Y' will by default not have been taken into account by the engine.

Reasoning on ontology data is the act of deriving information that is not explicitly expressed in the ontology, e.g. deriving inverse relationships. If a reasoner is applied on a SPARQL engine, the reasoner will create inferred data, i.e. the derived data.

An active reasoner might considerably slow down a SPARQL engine. In addition, since not all reasoners produce the same inferred data, one can choose to use static inferred data. This means that inferred data on a given ontology is generated once (optionally with multiple engines). This data can then be verified and included in the original ontology. Doing so will reduce the execution time of a query. The biggest disadvantage is that it is harder to modify the ontology: some information might be duplicated in different expressions, and the ontology might lack inferred data when adding new expressions. The size of the ontology also increases considerably. In this master dissertation however, the method of including inferred data in the original ontology was preferred since the main purpose was to research possibilities with the ontology and with a tool for retrieving patterns — more than researching live reasoners, which is rather a possible discussion point for future work on this subject.

4.5 Conclusion

This chapter described how the ontology can be used to query information about patterns. Examples of queries were given to illustrate the possibilities. Assumptions on the used technology (e.g. SPARQL) and software will be discussed in detail in Chapter 6.

Chapter 5

Querying tool

5.1 Introduction

In Chapter 3 and 4, an explanation was given on the ontology design and the ways to query the ontology. Software architects however do not always have the knowledge to work with querying languages like SPARQL. Both a visualisation of the search results and a user-friendly tool for defining queries can decrease the amount of time needed to find relevant patterns. In this chapter, a web-based application is presented to serve this purpose. A more user-friendly manual of the tool is given in Appendix A, which leaves out the technical details.

5.2 Application features

A selection of features was made based on the need of architects and generalisation of existing categorisation methods found in literature (see Chapter 1).

5.2.1 Pattern look-up

The homepage of the web-tool allows users to look up patterns based on categorisation criteria. Figure 5.1 shows a screenshot of this page.

In the left panel, the most important categorisation types are listed as seen in Chapter 2:

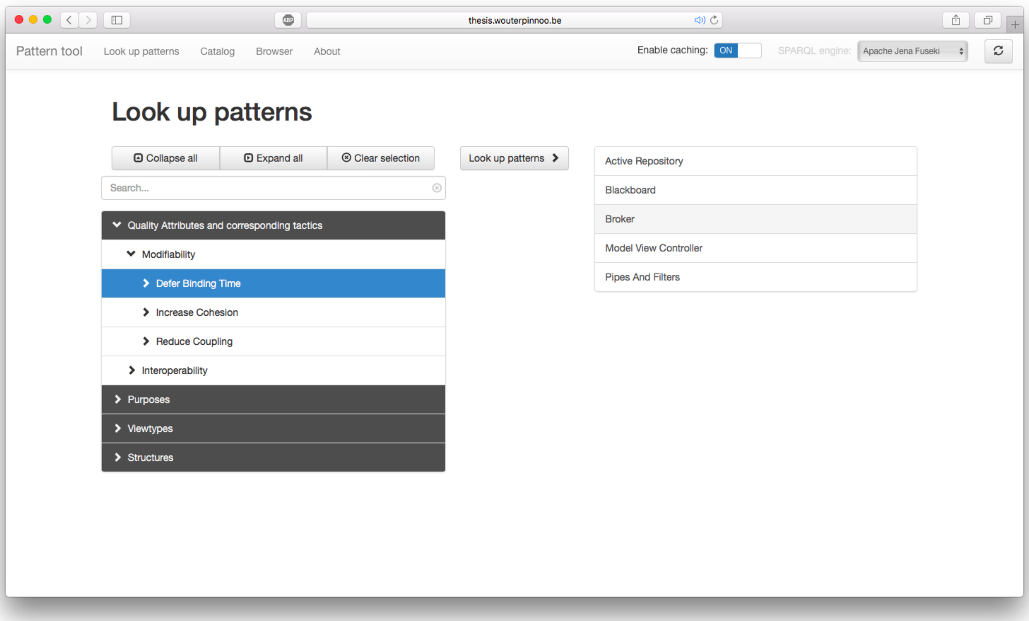


Figure 5.1: Homepage of the web-tool: pattern look-up

1. **Quality Attributes**: quality attributes as main category with as direct subcategories those tactics that achieve this quality attribute. Further categorisation is made with tactics that are marked as children of other tactics.

2. **Purposes**: categorisation of the goals of patterns. A goal can for example be "a problem that has to be tackled by the pattern" or "an area wherein the pattern wants to spend its focus on". Purposes are similar to APs, which were found to be a useful method for categorisation of patterns in Chapter 1. Purposes are categorised and are given a name. For example, all purposes that have the common property to design a distributed system are categorised in the *Distributed System* purpose. More details about a purpose can be viewed on the *Catalog Item Details* page, as will be discussed in Section 5.2.3. For example, if a description of the purpose is defined in the ontology, it will be shown on that page.

3. **Viewtypes**: categorisation based on which viewtype the patterns use.

4. **Structures**: categorisation made on the internal structure of patterns. Some patterns seem to have very similar internal structures. These structures are categorised using common properties. For example, the patterns *Publisher-Subscriber* and *Client-Server* have the common property to have independent

components. Therefore, the structure category *Independent Components* is listed on the home page. The internal structures of these patterns will differ from e.g. patterns using repositories-like internal structures.

These categorisation types are ordered according to the relevance in searching architectural patterns. Quality attributes are in most cases the most relevant measures since architects are looking for patterns that can tackle real-world problems, like environments with lack of certain quality measures.

The list of categorisation types is a collapsible tree, as depicted in detail in Figure 5.2. Hierarchy in the tree represents inheritance of objects in the ontology through the `isChildOf` and `isParentOf` relationships. In Figure 5.2 the selection interaction is also depicted. The user can select multiple items from the hierarchy, which appear in blue. For example, the user can select both a parent item in the tactics hierarchy (which will also select all child tactics) and an item in the purposes hierarchy to perform more a specific search in the patterns.

The user can also perform manual search in the hierarchy, as shown in Figure 5.3. Selecting an item in the search results will automatically select the respective items in the hierarchy. After the user has made his selection, the web tool will list all relevant patterns in the right panel, as seen in Figure 5.1. When clicking on a pattern, the user will be redirected to the *Catalog Item* page. The details of this page will be given in Section 5.2.3.

5.2.2 Catalog

On the catalog page, the web-tool retrieves a list of all objects of all types: *patterns*, *structures*, *tactics*, *quality attributes*, *purposes*, *views*, *viewpoints* and *viewtypes*. These lists are ordered in tabs, as depicted in Figure 5.4. Again, clicking on one of the items will redirect to the *Catalog Item* page.

In each tab, the user can filter the list with the search field on the top. Figure 5.5 depicts this feature.

In case the Stardog engine is selected, the catalog provides a method to add new objects. Figure 5.6 illustrates this feature. Depending on which tab is active (patterns, structures, ...), the dialog for adding items is adjusted to use the relevant relationships.

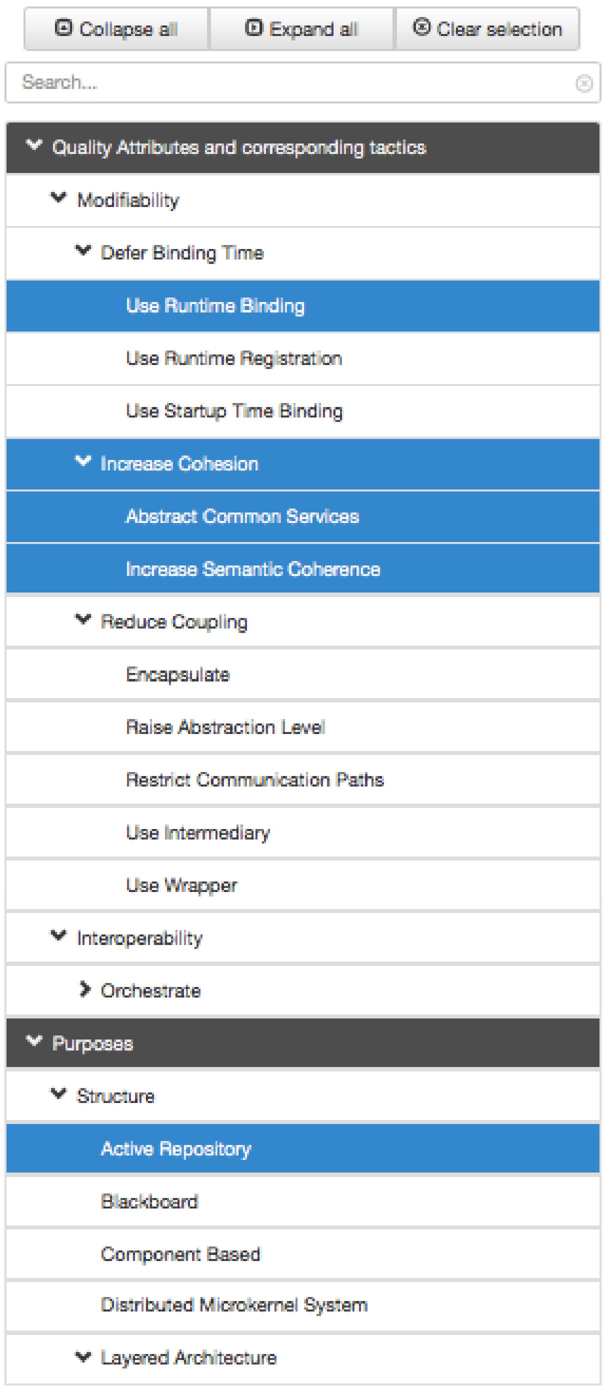
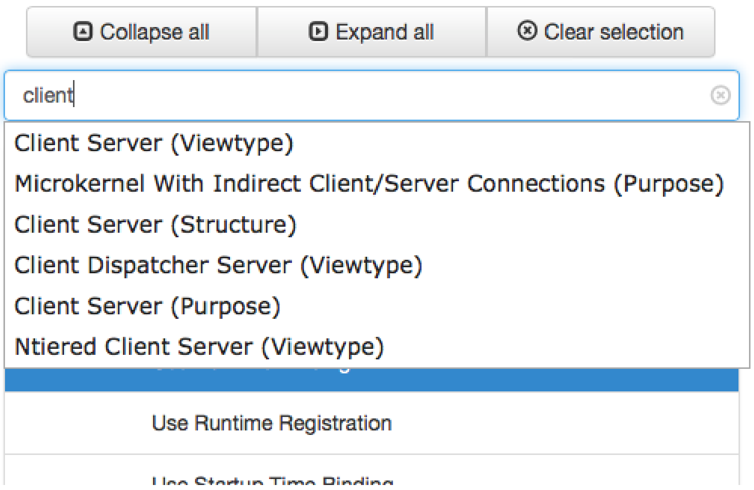


Figure 5.2: Detail view of the list of categorisation types



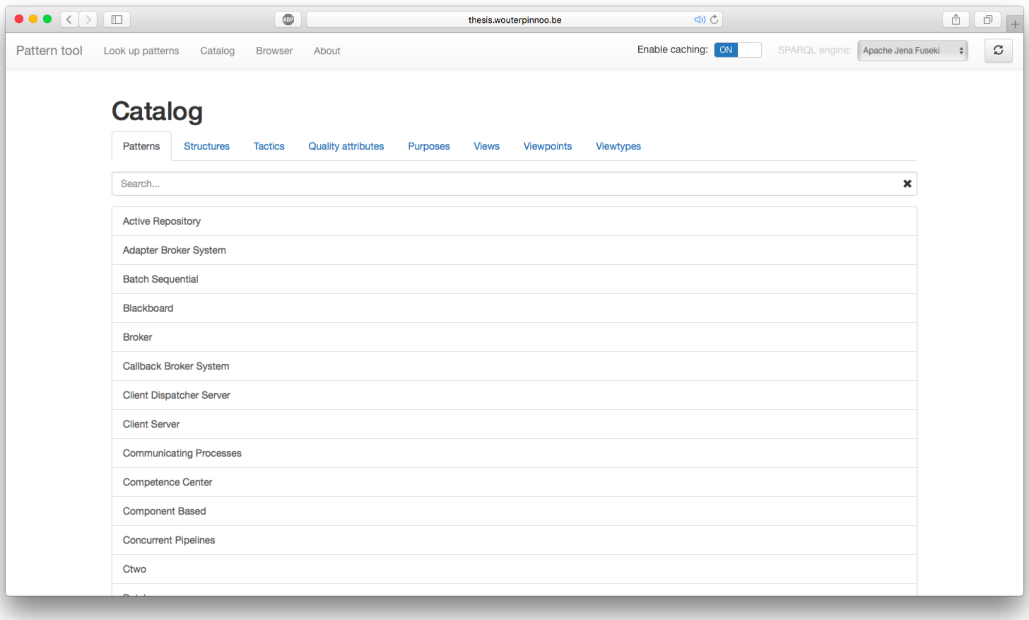
client

Client Server (Viewtype)
Microkernel With Indirect Client/Server Connections (Purpose)
Client Server (Structure)
Client Dispatcher Server (Viewtype)
Client Server (Purpose)
Ntiered Client Server (Viewtype)

Use Runtime Registration

Use Startup Time Binding

Figure 5.3: Searching in the list of categorisation types



Enable caching: ON

SPARQL engine: Apache Jena Fuseki

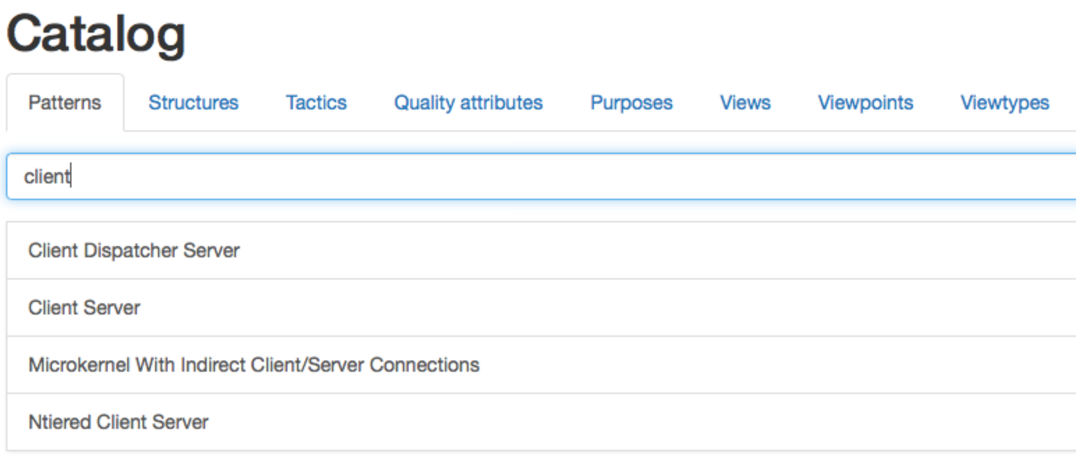
Catalog

Patterns | Structures | Tactics | Quality attributes | Purposes | Views | Viewpoints | Viewtypes

Search...

Active Repository
Adapter Broker System
Batch Sequential
Blackboard
Broker
Callback Broker System
Client Dispatcher Server
Client Server
Communicating Processes
Competence Center
Component Based
Concurrent Pipelines
Ctwo

Figure 5.4: Catalog list



Client Dispatcher Server
Client Server
Microkernel With Indirect Client/Server Connections
Ntiered Client Server

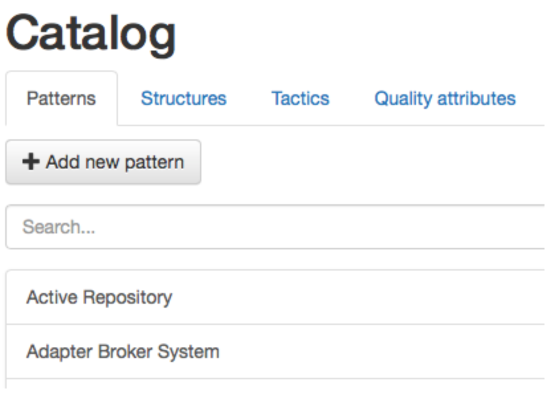
Figure 5.5: Filtering the catalog list

5.2.3 Catalog item details

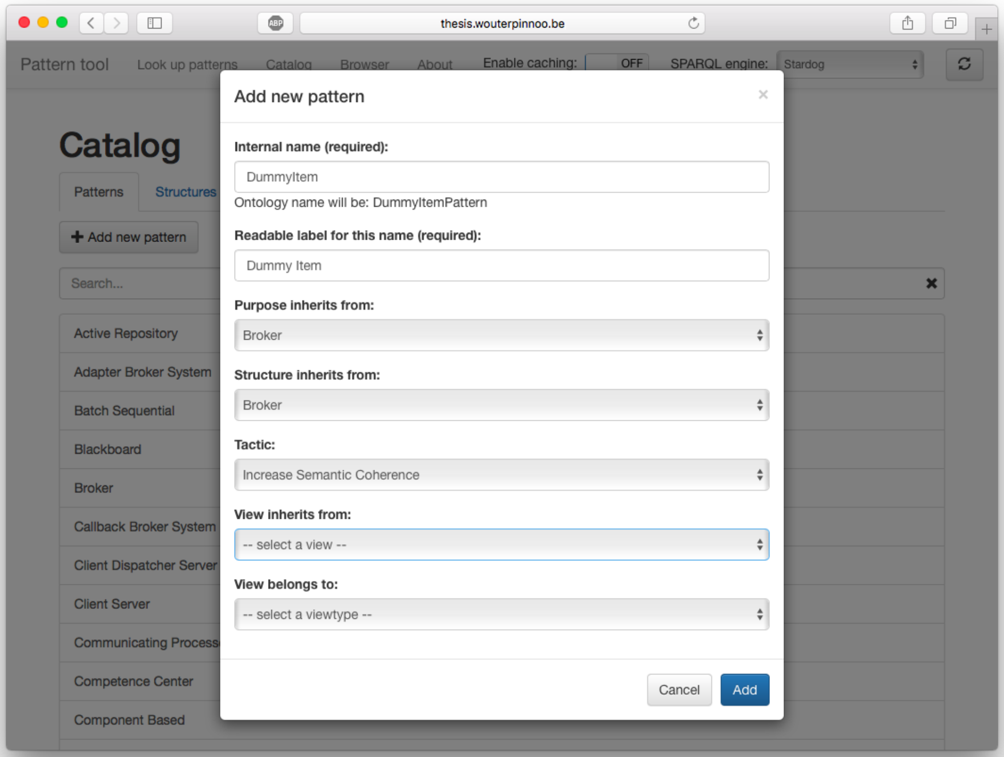
The *Catalog Item* page lists details on any object. The type of the object is recognised in the second-last segment of the URL. The last segment indicates the ontology URL of the respective object. E.g., details on the *Broker* pattern are retrieved on the URL `<domain>/pattern/BrokerPattern/` while the tactic *Increase Semantic Coherence* is retrieved on `<domain>/tactic/IncreaseSemanticCoherence/`.

The main purpose of this page is to provide the architect with *all* relevant information about the queried object. All relationships that can be related to the object are queried in the ontology. Also, all links to external resources are retrieved. As discussed in Section 3.3, resources are included in the ontology with annotations on either relationships between objects or on objects directly. With every object or relationship retrieved on this page, the ontology is queried for annotations (on those relationships) that contain resource information. Those resources are included in columns next to the related object.

Figure 5.7 depicts this page. In the first section, related resources of the object are listed. If one of those resources is a Wikipedia abstract (provided through the DBpedia ontology), the abstract is shown on the top of the page along with a hyperlink to the Wikipedia page. In the following parts of the page, all related relationships are listed along with links to resources that define that relationship. If an ISBN number was provided in the ontology, the tool will also show a hyperlink to the Google Books page of that ISBN number.



(a) Button for adding items — only active when using the Stardog engine



Add new pattern

Internal name (required): DummyItem

Ontology name will be: DummyItemPattern

Readable label for this name (required): Dummy Item

Purpose inherits from: Broker

Structure inherits from: Broker

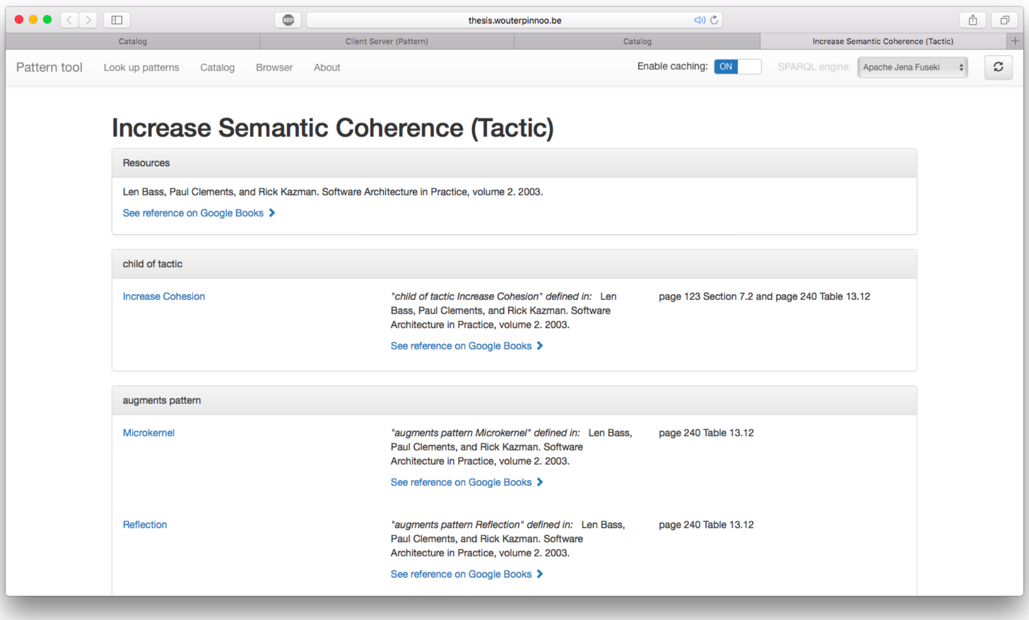
Tactic: Increase Semantic Coherence

View inherits from: -- select a view --

View belongs to: -- select a viewtype --

(b) Dialog for adding objects to the ontology

Figure 5.6: Add item to the catalog



Enable caching: ON

SPARQL engine: Apache Jena Fuseki

Increase Semantic Coherence (Tactic)

Resources

Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, volume 2. 2003.

See reference on Google Books

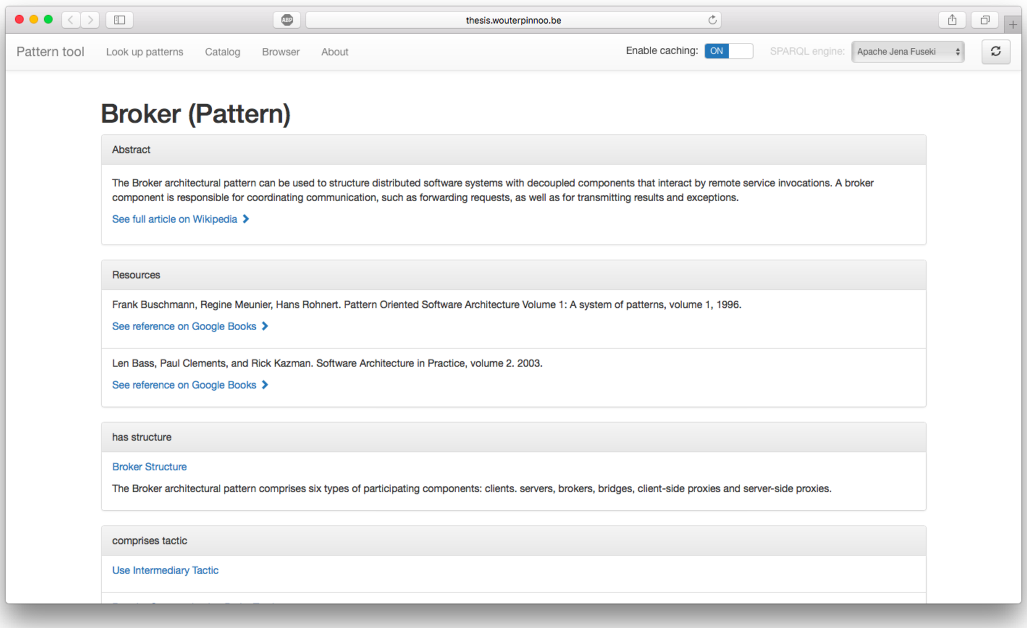
child of tactic

Increase Cohesion	*"child of tactic Increase Cohesion" defined in:* Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, volume 2. 2003. See reference on Google Books	page 123 Section 7.2 and page 240 Table 13.12

augments pattern

Microkernel	*"augments pattern Microkernel" defined in:* Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, volume 2. 2003. See reference on Google Books	page 240 Table 13.12
Reflection	*"augments pattern Reflection" defined in:* Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, volume 2. 2003. See reference on Google Books	page 240 Table 13.12

(a) Item with details of resource links (e.g. page numbers) provided



Enable caching: ON

SPARQL engine: Apache Jena Fuseki

Broker (Pattern)

Abstract

The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

See full article on Wikipedia

Resources

Frank Buschmann, Regine Meunier, Hans Rohnert. Pattern Oriented Software Architecture Volume 1: A system of patterns, volume 1, 1996.

See reference on Google Books

Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, volume 2. 2003.

See reference on Google Books

has structure

Broker Structure

The Broker architectural pattern comprises six types of participating components: clients. servers, brokers, bridges, client-side proxies and server-side proxies.

comprises tactic

Use Intermediary Tactic

(b) Item with a wikpedia abstract provided

Figure 5.7: Catalog item details

5.2.4 Browser

The browser page is meant for quickly navigating through patterns and related concepts. This need was identified when trying on my own to find a relevant pattern in a given context. It was noticed that it took quite a long time to find what was needed, since every step in the navigation from a quality attribute to the needed pattern took a long time: every page load, even with server-side caching (see Section 5.2.5.1),took too much time to be user-friendly. Therefore a more user-friendly method of browsing was designed. The result is depicted on Figure 5.8.

On this page, an infinite long interactive tree is shown. The root node has four children: the four categorisation types *quality attributes with corresponding tactics*, *purposes*, *structures* and *viewtypes*. When clicking on one of the nodes, the same children appear as on the homepage. However, when reaching a node that has no children on the homepage, one can still navigate further. A click will trigger a dynamic load of content, i.e. loading the yet to retrieve data via an API call with Javascript. Patterns, for example, are leaf nodes in the hierarchy on the homepage. When clicking on a pattern on this page, all relationships are added as children (except those relationships that don't are applicable for this pattern, or relationships that don't have any objects to target to). Clicking on such a relationship will reveal all objects related to the previous object through the clicked relationship. From this node, the same procedure could be followed again. This way this browsing tree has infinite length, and a user can browse quickly through objects and their related objects.

5.2.5 User preferences

Following user settings were initially set-up in the source code for debugging purposes and were afterwards integrated in the user interface to aid architects with experimenting with this web tool.

5.2.5.1 Server-side caching

Every time a page on this web tool loads, several SPARQL queries are executed. On the home page for example, five queries are executed: one for each mentioned

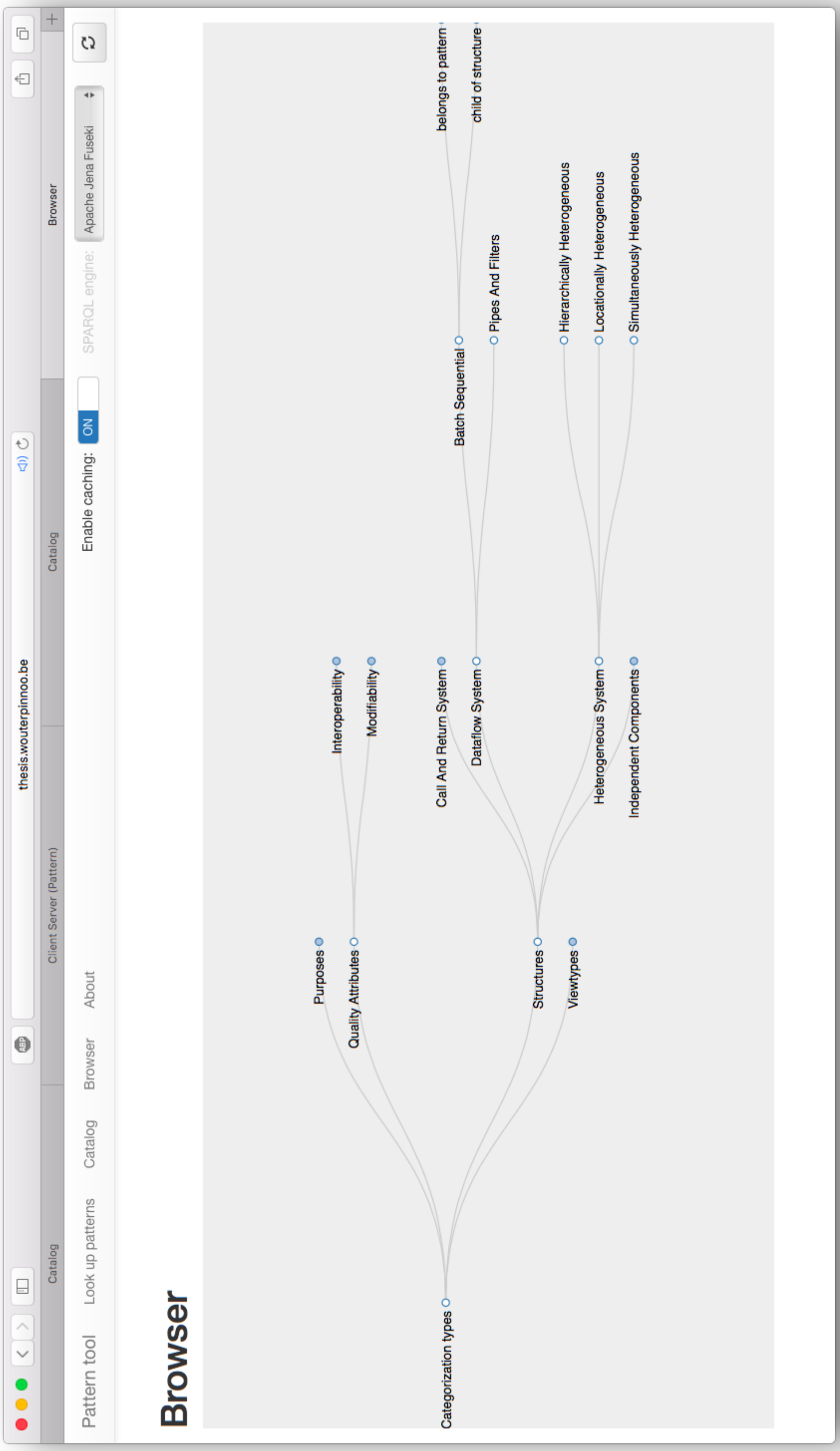
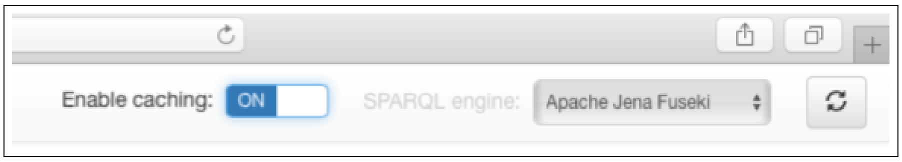
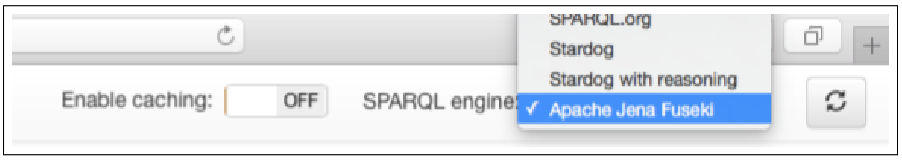


Figure 5.8: Browser page



(a) with caching enabled (engine switch disabled)



(b) with caching disabled

Figure 5.9: Screenshot of the user preferences options

categorisation type and one for the quality attributes that are achieved by the tactics. The total execution time of these queries is on average 1 to 1.5 seconds, and the generated output file (as will be discussed in Section 6.3) has a significant size. This can make the web tool less user-friendly. Therefore, server-side caching was implemented, so that the generated data on every page is saved. The only execution time needed on every page load is the rendering of the HTML documents as there are no SPARQL queries to be executed. Evidently, this functionality can only be used if there is no data added to the ontology while using the web tool. When the server-side caching is enabled, the functionality to select SPARQL engines is automatically disabled (see Section 5.2.5.2). Figure 5.9 illustrates this feature in the web tool. In Section 6.2.4, an example is given of the implementation of the cache.

5.2.5.2 Select preferred SPARQL engine

The web tool is implemented to use several SPARQL engines. The conversion of SPARQL queries into HTTP requests, and the conversion of their HTTP responses into readable JSON data is made independent of the used SPARQL engine. This makes it possible to provide functionality for the user to switch SPARQL engines. Figure 5.9 illustrates the use of this option. When an option is selected, the cache is invalidated and the page is reloaded with the selection engine. If an option other than *Stardog* is selected, the web tool will hide all functionality related to adding data to the ontology.

5.3 Conclusion

This chapter proposed a prototype of a visualisation of the information that is contained in the ontology. The visualisation was designed to fit the needs of the architects: browsing through relevant pattern information without having to write queries on the ontology.

The architecture of the tool will be discussed in Chapter 6 along with technical decisions.

Chapter 6

Architecture and technical decisions

In Chapter 5, a tool was described to query the ontology in a user-friendly way. The architecture of this tool, along with technical decisions, is discussed in this chapter.

The purpose of this tool is that architects can easily access a visualisation tool and find relevant patterns in a fast way. To overcome problems like software compatibility, maintenance, package dependencies and installation time, a web tool is chosen over a native tool. Other benefits of using a web tool are the ease of making visualisations — which are in general slightly more complex to implement in native tools — and the independence of the used operating system.

As a framework for the web tool, the Flask [14] micro-framework is presumed to be the best suitable solution. The computation complexity of this tool is rather low and the architecture of the software in this web tool is not complex, which eliminates the need for a big framework. Micro-frameworks are easily extendible with other micro-frameworks and provide fast APIs with little overhead. The choice for a Python framework — and more specifically the Flask framework — was made with code complexity in mind. Python has great libraries for JSON manipulation operations and HTTP requests that require a small amount of code. Being able to have a web tool with limited code complexity benefits modifiability: developers can more easily find their way in the code to extend the tool to their needs or the needs of the software architects in their company.

With modifiability in mind, the front-end of the web tool is chosen to be completely separated from the query processor. The web tool, implemented with Flask, will only handle the conversion of query results into a JSON format that can be used in visualisation. Thus, the three main components of the web tool are:

- the SPARQL endpoint: a server-side engine that does the query processing on the ontology;
- the Flask web tool: handles API requests and does conversion of data from the SPARQL engine to a JSON format suitable for visualisation;
- a Javascript front-end visualisation.

Figure 6.1 illustrates these components, which will be described in detail in the following sections. The interfaces from the web tool to other components in this architecture are (indicated in blue):

1. With the Fuseki and SPARQL.org engines: the web tool sends a SPARQL query with a HTTP POST call and gets a JSON document in return containing the results. The exact interface format of these engines differ from those of interfaces 3 and 4.
2. With the Google Books API: the web tool sends an ISBN number with a HTTP GET call and gets information about the corresponding book in return (a JSON document).
3. With the DBpedia SPARQL engine: the web tool sends a SPARQL query with a HTTP POST call to the DBpedia endpoint and gets the result in return in a JSON document.
4. With the Stardog SPARQL engine: similar to the other SPARQL engines, the Stardog engine can be queried with a HTTP POST call.
5. The front-end visualisation queries data from the web tool. The queried data is returned in JSON documents.

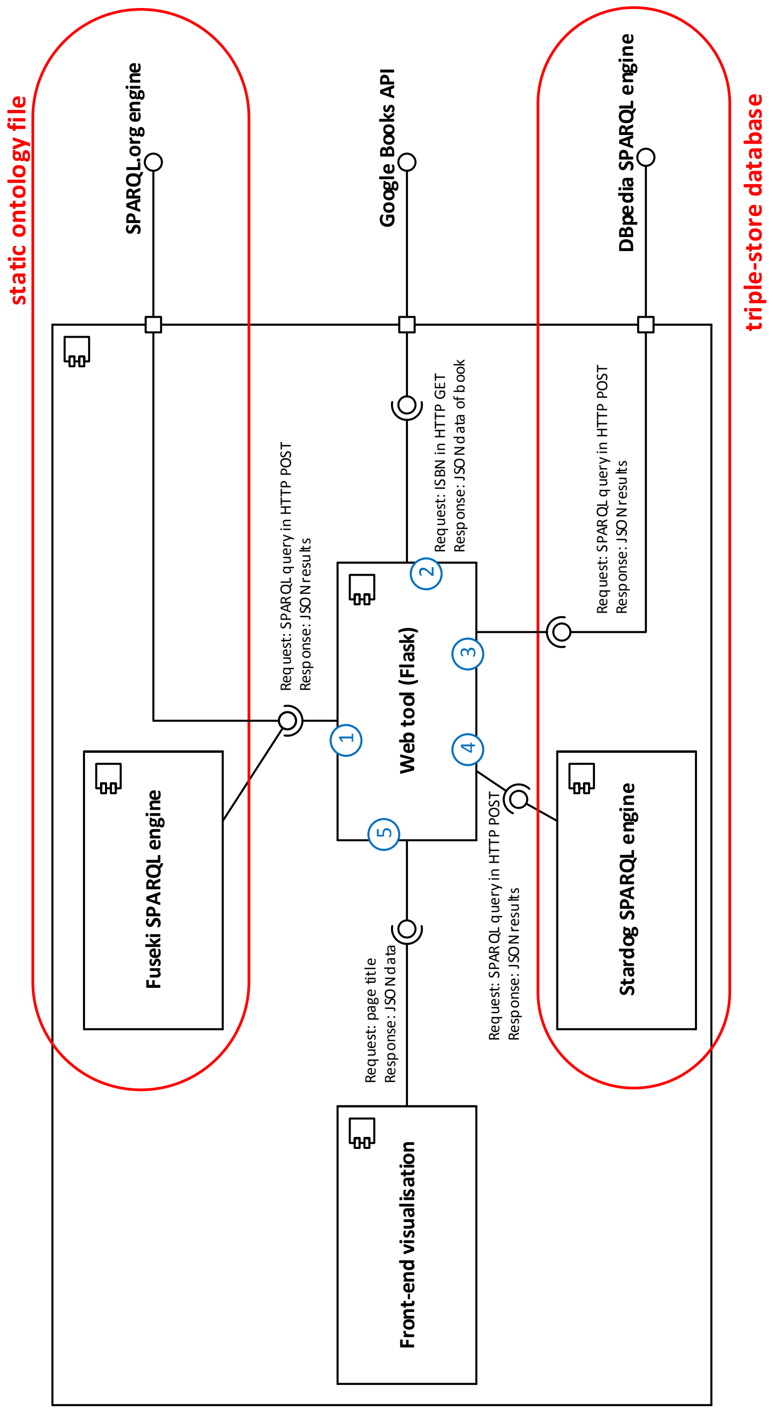


Figure 6.1: Components of the webtool. The indications in red denote the type of SPARQL engines. Interfaces are numbered in blue.

6.1 SPARQL engines

A SPARQL engine is responsible for generating results for SPARQL queries on a given ontology. The two main methods for storing the ontology on which queries must be executed, also indicated in red on Figure 6.1, are:

1. hosting a static ontology file;
2. importing an ontology file into a triple-store database.

These methods are discussed in Sections 6.1.1 and 6.1.2. A conclusion and comparison of the SPARQL engines is made in Section 6.1.4.

6.1.1 Engines with static ontology file

In order to provide the ontology part of this system as an abstraction to the visualisation part and the Flask web tool, the Apache Jena framework was chosen in combination with the Fuseki interface [2]. Apache Jena is a framework that allows services to query on RDF[1] data [47] and to modify the RDF data. In this case, the RDF data is simply the OWL ontology file. The Fuseki interface allows the Jena framework to be accessed with HTTP calls. These frameworks were chosen because they were easy to set up in this environment.

The Fuseki server is a stand-alone service that provides an HTTP API on the `<domain>/sparql` endpoint. A SPARQL query can be send to this endpoint in a HTTP POST request with the `query` parameter. In this request, the location of the ontology file must be provided with the `default-graph-uri` parameter. Fuseki will load the ontology file and execute the query in-memory.

Figure 6.2 depicts the information flow with the use of the Fuseki interface. One submits an ontology file (in this case a link to an ontology file) along with the SPARQL query. The Fuseki interface will load the contents of this ontology file and convert it to a RDF dataset that was set up when setting up Fuseki. The RDF dataset is an in-memory, non-persistent storage. Fuseki passes the RDF dataset and the SPARQL query to Jena, which generates the SPARQL results as RDF triples.

[1]RDF is a semantic language [47] used to represent data in graph triples, usually in the form *(node, edge, node)* for the expression *subject $\rightarrow$ has something to do with $\rightarrow$ object*.

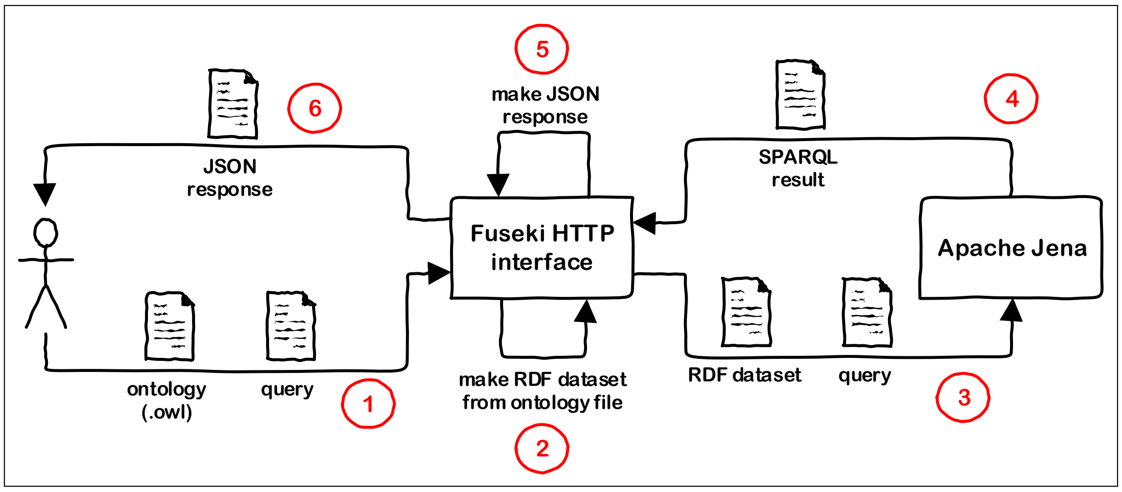


Figure 6.2: Interaction with the Fuseki HTTP interface

Fuseki converts those triples into a readable JSON format and returns it in the HTTP response.

Listing 6.1 shows a SPARQL query made through the Fuseki HTTP interface, written in Python code with the *requests* library, along with its output. In this query, all Quality Attributes are queried,

?object a :QualityAttribute

along with their labels,

?object rdfs:label ?label.

Refer to Chapter 4 for details on the format of this query.

```
> python
Python 2.7.10 (default, Sep 23 2015, 04:34:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.72)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests, json
>>> data = {
...     'output': 'json',
...     'default-graph-uri': 'http://thesis.wouterpinnoo.be/patterns.owl',
...     'query': ('PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> '
...               'PREFIX owl: <http://www.w3.org/2002/07/owl#> '
...               'PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> '
...               'PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> '
...               'PREFIX : <http://thesis.wouterpinnoo.be/patterns.owl#> '
...               '    SELECT ?object ?label'
...               '    WHERE { '
...               '        ?object a :QualityAttribute. '
...                 '        ?object rdfs:label ?label . '
...                 '    } '
...                 '    ORDER BY ?object')
... }
>>> request = requests.post('http://thesis.wouterpinnoo.be/fuseki/sparql', data=data)
>>> print json.dumps(request.json(), indent=4)
{
    "head": {
        "vars": [
            "object",
            "label"
        ]
    },
    "results": {
        "bindings": [
            {
                "object": {
                    "type": "uri",
                    "value": "http://thesis.wouterpinnoo.be/patterns.owl
                              #InteroperabilityQualityAttribute"
                },
                "label": {
                    "type": "literal",
                    "value": "Interoperability"
                }
            },
            {
                "object": {
                    "type": "uri",
                    "value": "http://thesis.wouterpinnoo.be/patterns.owl
                              #ModifiabilityQualityAttribute"
                },
                "label": {
                    "type": "literal",
                    "value": "Modifiability"
                }
            }
        ]
    }
}
>>>
```

Listing 6.1: Execution of a SPARQL request with HTTP

As seen in Figure 6.1, the SPARQL.org engine is also used in this project. The previously discussed Fuseki engine runs on the same server as the web-tool, while SPARQL.org is another, external, running instance of Fuseki that was deployed with probably different configuration parameters. During the testing of the web tool, several SPARQL engines were tested to verify the compatibility with external services. During these tests, minor differences were noticed in the extent to which certain triples were returned in the output. Presumably this can be caused by the configuration of a different reasoner in SPARQL.org.

6.1.2 Engines with triple-store database

Triple-store databases store RDF triples in a persistent way and are loaded at installation time, rather than loading an ontology file at execution time. The persistent nature of the storage will make it possible to not only query the data, but also make modifications in it. In fact, all CRUD (Create, Read, Update, Delete) operations are supported in the triple-store that will be used. Since the ontology that was proposed in Chapter 3 is written in the OWL language, the ontology is first manually converted into a Turtle syntax (as a `.ttl` file). The Turtle syntax makes use of the RDF data model to store expressions as triples. This was done with an internal tool of the Protégé software tool. Listing 6.2 illustrates the format of a `.ttl` file. In this example, we can distinguish three triples, one of which is a combined triple: the first triple states that `BrokerImplStructure` is both a `Structure` and a `NamedIndividual`. The second and third triple respectively assign a label and a `belongsTo` relationship to the `BrokerImplStructure` individual.

```
root@web:~/stardog/bin# sed -n '968,977p' /root/patterns.ttl

ns3:BrokerImplStructure rdf:type ns3:Structure ,
                                 ns1:NamedIndividual ;

                        ns2:label "Broker" ;

                        ns3:belongsTo ns3:BrokerPattern .
```

Listing 6.2: Turtle (.ttl) syntax

Next, a Stardog server [11] was set up: a SPARQL engine that uses a triple-store database. A new database was created and the triples were loaded in the database. From that moment on, the ontology content could be queried or modified using SPARQL queries directly on the triple-store database. Listing 6.3 illustrates the process of creating the triple-store database, inserting data and querying data.

```
root@web:~/stardog/bin# ./stardog-admin db create -n testDatabase /root/patterns.ttl
Bulk loading data to new database testDatabase.
```

```
Loaded 4,393 triples to testDatabase from 1 file(s) in 00:00:06.594 @ 0.7K
    triples/sec.
Successfully created database 'testDatabase'.

root@web:~/stardog/bin# ./stardog query testDatabase "\
> INSERT DATA { \
>     :TestTactic rdf:type owl:NamedIndividual, :Tactic . \
>     :TestTactic rdfs:label 'Test' . \
>     :TestTactic :augments :BrokerPattern .
> }"
Update query processed successfully in 00:00:00.166.

root@web:~/stardog/bin# ./stardog query testDatabase "\
SELECT DISTINCT ?tactic ?label \
WHERE { \
    ?tactic a :Tactic . \
    ?tactic :augments [a :Pattern] . \
    ?tactic rdfs:label ?label . \
}"
+-----------------------------------+-------------------------------+
|              tactic               |             label             |
+-----------------------------------+-------------------------------+
| :RaiseAbstractionLevelTactic      | "Raise Abstraction Level"     |
| :IncreaseSemanticCoherenceTactic  | "Increase Semantic Coherence" |
| :UseRuntimeRegistrationTactic     | "Use Runtime Registration"    |
| :UseRuntimeBindingTactic          | "Use Runtime Binding"         |
| :EncapsulateTactic                | "Encapsulate"                 |
| :AbstractCommonServicesTactic     | "Abstract Common Services"    |
| :UseIntermediaryTactic            | "Use Intermediary"            |
| :RestrictCommunicationPathsTactic | "Restrict Communication Paths" |
| :TestTactic                       | "Test"                        |
| :UseStartupTimeBindingTactic      | "Use Startup Time Binding"    |
| :OrchestrateTactic                | "Orchestrate"                 |
+-----------------------------------+-------------------------------+

Query returned 11 results in 00:00:00.163
```

Listing 6.3: Stardog setup

Fortunately, Stardog also provides a HTTP interface that can be used for sending SPARQL queries over HTTP and retrieving results in a JSON format that is identical to the format of Fuseki as seen in Listing 6.1.

The information flow with the usage of Stardog is depicted in Figure 6.3. First, a triple-store database is created and the `.ttl` file loaded into the database via the Stardog Command-line Interface (CLI). Next, one can send a SPARQL query over HTTP to the Stardog HTTP interface, which will query the triple-store database directly and parse the results in a JSON format before sending it back to the

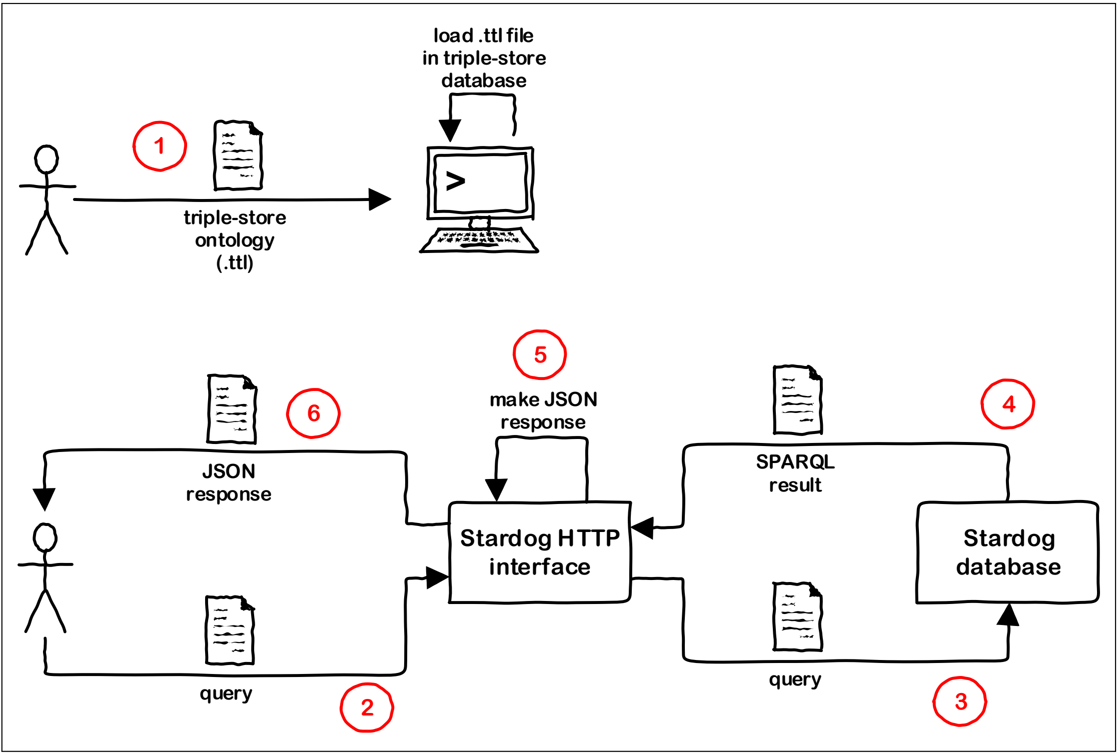


Figure 6.3: Interaction with the Stardog HTTP interface

requester.

Stardog also allows to quickly enable a reasoner on the triple-store database. As seen in Section 4.4, a reasoner generates on-the-fly inferred triples, given the relationships in the stored triples. Without reasoning, all possible transitive, reflexive, symmetric and inverse relationships must be explicitly defined. A reasoner can be enabled by adding the parameter `reasoner=true` in the HTTP request. It was noticed that enabling a reasoner on the current ontology increased the query execution time considerably.

6.1.3 DBpedia SPARQL engine

Components, individuals and relationships in the ontology can be annotated with links to external resources. Links to Wikipedia pages are annotated with the corresponding DBpedia URLs, as seen in Section 3.3. Those DBpedia URLs can easily be queried through a SPARQL engine hosted on *DBpedia.org*. In an identical fashion to the Fuseki SPARQL engine, the DBpedia engine provides a HTTP interface to send queries to, and returns results in a JSON format identical to the Fuseki

	Idle	Sequence of 5 queries (peak)
Fuseki	18 *MB*	91 *MB*
Stardog	179 *MB*	314 *MB*

Table 6.1: Memory consumption of SPARQL engines

JSON format. Querying the DBpedia ontology is always done with the DBpedia engine, while queries on the patterns ontology can be done with both the Fuseki, SPARQL.org and Stardog engines. The user can select his preferred engine in the web tool, as discussed in Section 5.2.5.2.

6.1.4 Conclusion and comparison of SPARQL engines

As discussed in the previous sections, the two most important methods to store data and perform queries on them are storing a static ontology file and storing triples in a triple-store database. In this section, a conclusion is made on some qualitative aspects of both methods.

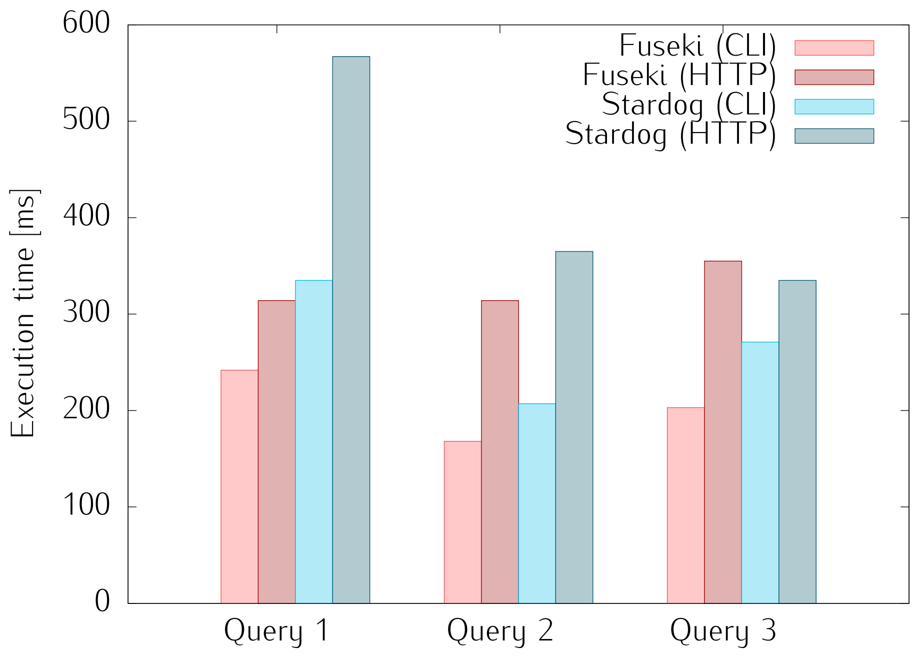
System and resource requirements Fuseki itself doesn't have specific system requirements since it is a stand-alone runnable that only serves as HTTP interface. The underlying core, Apache Jena, requires Java (any version). Stardog on the other hand relies on Java 8. The most significant difference between these two methods, is the memory consumption. Table 6.1 lists the memory footprint of both methods, in which we can see that the memory consumption of Stardog is notably higher than Fuseki.

Installation time Installation time of tools or frameworks is an important measure when the framework is not the core functionality in a project. In this master dissertation, several frameworks were tested in order to make conclusions on which framework is most suitable, so ease of installation was an important factor. Both frameworks — Fuseki and Stardog — are stand-alone packages that do not require any installations of dependencies. However, the usage of reasoning is far more easy to do with Stardog than with Fuseki. Stardog has already a built-in reasoner that can be enabled with a URL parameter `reasoning=true` in the HTTP request. Fuseki on the other hand requires a non-straightforward explicit configuration of the reasoner.

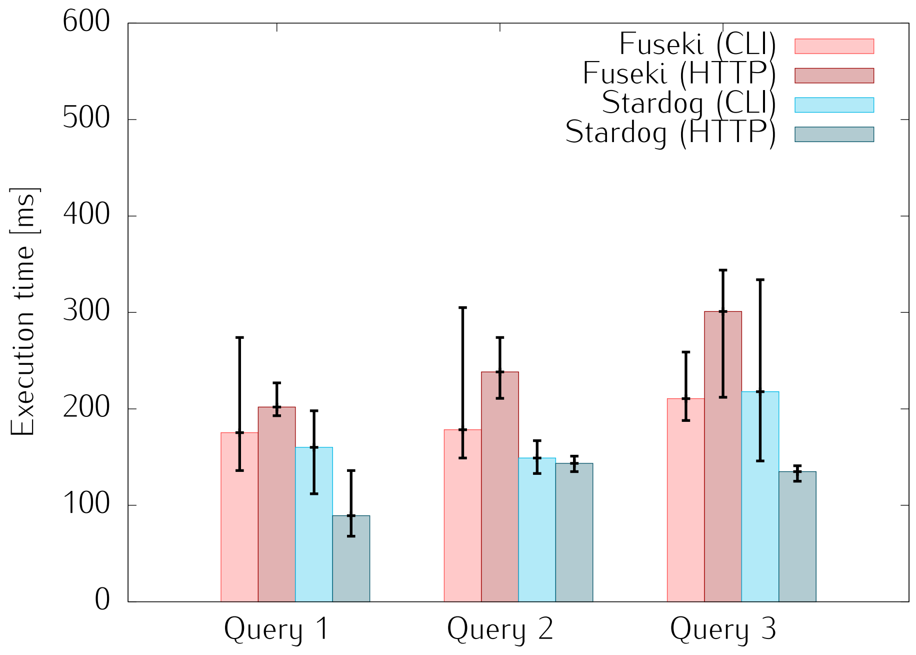
Query execution time As indicative measure for query execution time, the three first queries that are executed when loading the main page of the web tool were captured and used in this benchmark. The queries were executed multiple times on both the CLIs of the frameworks and their HTTP interfaces. With both the Fuseki and Stardog framework, reasoning was disabled. Instead, an inferred version of the data was used, as described in Section 4.4. A distinction was made between the first execution of each query and later executions since it was noted that the execution time significantly decreased after the first execution, due to caching.
Figure 6.4 depicts the results. The first graph, Figure 6.4a, visualises the execution time of the first execution of each query. The execution time of a query made on the HTTP interface is higher than that of the query made on the CLI, as expected. Indeed, the CLI directly executes queries on the core, while the HTTP interface is an additional abstraction layer that introduces some overhead. In most cases, except for the execution through the HTTP interfaces of query 3, the execution time with Stardog is higher than with Fuseki, suggesting that Fuseki is a more light-weight framework.
The second graph, Figure 6.4b, depicts the execution times after warm-up of the engines, i.e. after first executing queries on the engines without measuring their execution times. Error bars are included to indicate the range of the measurements. Two important differences with Figure 6.4a are notable:

- all measurements are significantly lower than the first execution, which suggests some caching in the core of the frameworks,
- in contrast to the Fuseki HTTP interface, the Stardog HTTP interface now allows for faster query execution than the Stardog CLI, which suggests that also the Stardog HTTP interface does some caching to improve the execution time.

Features In what was needed in this master dissertation, all used features from Fuseki are also included in Stardog. When looking at future functionality of the web



(a) Execution time of first execution



(b) Execution time after warmup

Figure 6.4: Execution time of different SPARQL engines

tool, Stardog provides more useful features. Stardog for example supports all Create, Read, Update, Delete (CRUD) operations, which allows the web tool to modify to the ontology.

Conclusion The frameworks Fuseki and Stardog were compared on a number of qualitative measures. The execution time of queries on the Stardog framework seems to be lower than on the Fuseki framework, except for queries that are executed for the first time. This is at the cost of more memory consumption, although the peak memory consumption is still acceptable.
Other benefits of Stardog are the ease of installation and the ability to support adding data to the ontology. This can be useful as the data of the web tool will be persistent. In case the web tool will be used to work on ontology files that the user can submit, it will be more difficult to use Stardog because Stardog requires that the ontology file is first converted into a Turtle format and then loaded in a triple-store database, as seen in Section 6.1.2. Fuseki on the other hand allows the execute queries directly on the ontology file.

6.2 Flask: API and conversion of data

As described in Section 6.1.1, both the Fuseki and Stardog interface provide HTTP access to the underlying SPARQL engine. The web tool, implemented with the Flask micro-framework, makes use of the HTTP interfaces to construct queries, execute them on the HTTP interfaces and convert them in a format that can be used in the visualisation component (see Section 6.3).

Table 6.2 lists the source files of the Flask web tool. Figure 6.5 depicts a simplified UML diagram of the classes. In the sections below, all components are discussed in detail.

6.2.1 Web instance

Web instance is a component that is implemented in `web.py`. In this component, all HTTP requests to the web tool are received. With Flask annotations, each request is mapped to a method in which first the corresponding `Page` classes are called to

Package	Files	Class name
pages	`abstract_categorisation_page.py`	AbstractCategorisationPage
	`abstract_item_page.py`	AbstractItemPage
	`abstract_page.py`	AbstractPage
	`add_item_api_page.py`	AddItemAPIPage
	`browser_api_page.py`	BrowserAPIPage
	`browser_page.py`	BrowserPage
	`catalog_item_page.py`	CatalogItemPage
	`catalog_page.py`	CatalogPage
	`look_up_patterns_api_page.py`	LookUpPatternsAPIPage
	`look_up_patterns_page.py`	LookUpPatternsPage
utils	`abstract_treenode.py`	AbstractTreeNode
	`browser_treenode.py`	BrowserTreeNode
	`lookup_patterns_treenode.py`	LookupPatternsTreeNode
	`request_util.py`	RequestUtil
	`tree_util.py`	TreeUtil
	`url_util.py`	UrlUtil
(none)	`constants.py`	(none)
	`web.py`	(none)

Table 6.2: Source files

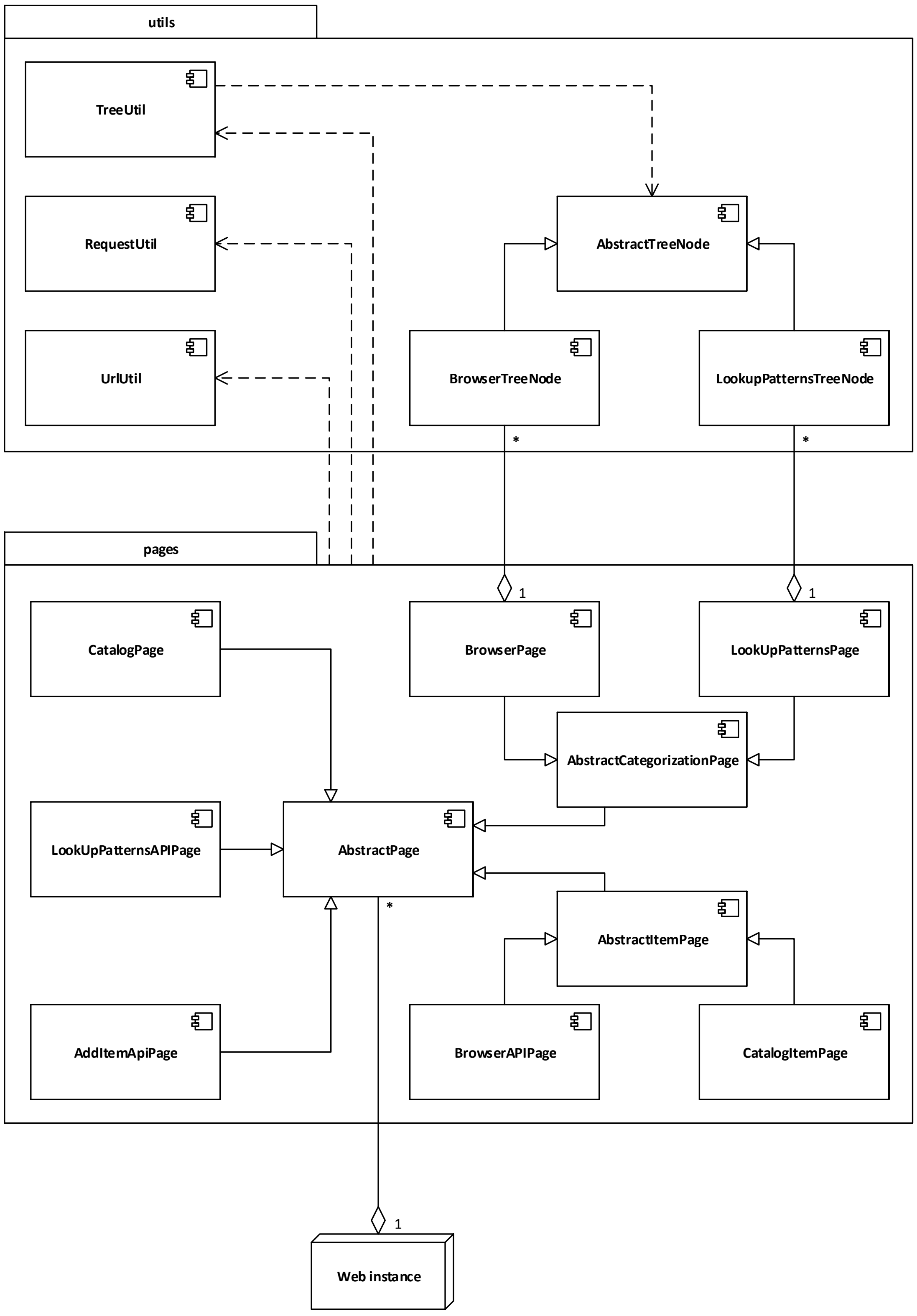


Figure 6.5: Simplified UML diagram of classes in the web tool

generate the necessary data, and next the actual HTML document is rendered using this data.

Listing 6.4 illustrates the routing of the URL that is used to show all details of an object, where an object can be a pattern, a tactic, etc. On the first line, the URL to be listened to is defined. The URL segments `<path:type>` and `<path:item>` indicate that this part of the URL can be variable. In this case, the `type` indicates the type of catalog item the user is requesting. This can be `pattern`, `tactic`, etc. This type is used in the visualisation to know what has to be shown: for patterns, the related structures, purposes, tactic, etc., are shown, while for tactics the related objects are patterns, quality attributes, viewpoints and views. The URL segment `item` indicates the exact name of this object in the ontology. This name will be used to construct the SPARQL queries.
In the method itself, a `CatalogItemPage` is made using these parameters and the resulting data is rendered in a Flask HTML document template.

```
@app.route('/catalog/<path:type>/<path:item>')
def catalog_item(type, item):
    global catalogItemPage

    catalogItemPage = CatalogItemPage(type, item)
    itemData = catalogItemPage.getData()

    return render('catalog_item.html',
                  title=itemData['title'],
                  itemData=itemData)
```

Listing 6.4: Routing of the catalog item page

6.2.2 AbstractPage

Pages are classes that, optionally given some parameters, construct a SPARQL query, execute it and parse the resulting JSON document into a JSON format that is readable by the visualisation component (see Section 6.3). `AbstractPage` is an abstraction for these classes that allows the *Web instance* component to retrieve the resulting data via the `getData()` method. The specific *Pages* are discussed below.

CatalogPage This page queries the ontology for all objects of any type and parses them in a JSON format that contains the type of the object, the ontology URL of that object, a readable name for that object and an URL to the page that contains all details of this object (implemented in *CatalogItemPage*, more details in Section 5.2.2).

LookUpPatternsAPIPage This is an API page, which means that is not an actual HTML document that will be rendered, but is instead a page that is callable through an API and returns a JSON document. This page is used on the home page of the web tool to retrieve patterns. After the user has selected categories in the left panel, the categories are send to this API page and the resulting patterns are returned in a JSON format to the home page, which appear in the right panel.

AddItemsApiPage Again, this page is an API page that returns a JSON document. This page is used in the dialog for adding items to the ontology. Contents of the new object are passed as arguments, the object is added to the ontology and the returned JSON document contains a flag that indicates whether the adding operation succeed or failed.

6.2.3 AbstractItemPage

`AbstractItemPage` is an abstraction for pages that show details of an object in the ontology. Queries for such pages are for the most part identical, but the format in which the responses of the queries are converted into can differ.

CatalogItemPage This page lists for a given object all related content that is known in the ontology. The JSON output format is chosen in such a way that rendering the content in a HTML document is kept simple.

BrowserAPIPage This API uses the same queries as `CatalogItemPage` but uses a different JSON output format. This API page is triggered when a user clicks on a node in the browser page that didn't previously contain any content. This page queries the ontology for all related content in the ontology, in such a way that new nodes can be added in the browser. The JSON output format is fixed, as required by the used library of the browser. See Section 6.3 for details on this.

6.2.4 AbstractCategorisationPage

This abstract page generalises functionality for pages that visualise categorisation types. Again, queries are for the most part identical but the output JSON format differs.

LookUpPatternsPage This is the homepage of the web-tool that allows users to retrieve patterns given some categories that are selected by the user. The queries for the contents of this page are static, i.e. the queries don't contain any unknown variables. Due to the static nature of this page, it can easily be cached on the server as discussed in Section 5.2.5.1. Listing 6.5 illustrates how caching is implemented for this page.

```
lookUpPatternsPage = None                               # cache page on the server
cacheEnabled = True

def isRefreshNeeded(pageObj):                           # utility function
    return not cacheEnabled \
           or pageObj is None \
           or request.args.get('refresh') is not None

@app.route('/')                                         # execute this method when
def lookUpPatterns():                                   # navigating to <domain>'/'
    global lookUpPatternsPage
    if isRefreshNeeded(lookUpPatternsPage):             # should cache be used?
        lookUpPatternsPage = LookUpPatternsPage()
    categorisations, tags = lookUpPatternsPage.getData()  # retrieve actual content
    return render('look_up_patterns.html',              # render content
                  categorisations=categorisations,
                  tags=tags)
```

Listing 6.5: Implementation of the cache of the LookUpPatternsPage

The output JSON format is chosen to fit the required format by the used library of the hierarchical tree on the homepage. See Section 6.3 for more details on this.

6.2.5 TreeNode classes

Tree nodes are used in both the visualisation on the home page (the hierarchical tree) and the browser page. The `AbstractTreeNode` class — and the implementation classes — are served as utility classes for the generation of the JSON documents required in both pages. The main differences between `BrowserTreeNode` and `LookupPatternsTreeNode` is the required format by the visualisation libraries on the respective pages.

6.2.6 Utility classes

Three utility classes are used:

- `RequestUtil`: contains methods for executing the actual SPARQL queries.
- `UrlUtil`: contains an utility method for retrieving the last segment of an URL.
- `TreeUtil`: utility functions operating on tree nodes. For example, the JSON output format of SPARQL is a flat array, while the visualisation library of both the homepage and the browser page require a hierarchical format. A conversion method is implemented in this class. The input of this method is a flat JSON array. Assume as example objects `A`, `B`, and `C`, with `C` a child object of `B`. The SPARQL query response will look like this:

```
{
    "head": {
        "vars": [
            "parent",
            "child"
        ]
    },
    "results": {
        "bindings": [
            {
                "parent": {
                    "type": "uri",
                    "value": "<root>"
                },
                "child": {
                    "type": "uri",
                    "value": "<url>/patterns.owl#A"
                }
            },
            {
                "parent": {
                    "type": "uri",
                    "value": "<root>"
                },
                "child": {
                    "type": "uri",
                    "value": "<url>/patterns.owl#B"
                }
            },
            {
                "parent": {
                    "type": "uri",
                    "value": "<url>/patterns.owl#B"
                },
                "child": {
                    "type": "uri",
                    "value": "<url>/patterns.owl#C"
                }
            }
        ]
    }
}
```

This format is converted to something like:

```
{
    "nodes": [
        {
            "name": "A",
            "children": []
        },
        {
            "name": "B",
            "children: [
                {
                    "name": "C"
                }
            ]
        }
    ]
}
```

6.3 Visualisation

The visualisation component is developed as independent from the Flask web tool as possible. The interaction between the visualisation component and the web-tool is done through JSON documents that are easily usable in other visualisations. The main components in the visualisation are:

- **Home page: pattern look-up:** A hierarchical tree is used on the home page to visualise the categorisation tree. This is implemented with the Bootstrap-Treeview library [29]. When clicking the *Look up patterns* button, patterns are retrieved via an Ajax call and parsed into a Bootstrap Listview.
- **Catalog (-item) page:** The catalog page and the page with the contents of a catalog item are rendered using standard Bootstrap components.
- **Browser:** The browser page is implemented with *Collapsible Tree* from the d3.js examples library.

Listing 6.6 shows the JSON document that is generated when querying the details page of the *Broker* pattern. Because of the size of the JSON document, some irrelevant parts are discarded. This JSON document contains all information that can be shown in a visualisation.

In the first part, the name of the page, the title and a reference to the exact ontology URI are listed. Next, all information that can be shown in the *Wiki* part is listed. The *Wiki* part of the page is the uppermost part that contains (optionally) a Wikipedia abstract and references to resources that describe the queried item. In this case, two resources are listed along with their respective Google Books URIs.

Finally, the JSON document contains a list of all the relationships in which the queried object is participating. For example, the Broker pattern takes part in the *comprises tactic* relationship two times: with the *Use Intermediary* tactic and the *Restrict Communication Paths* tactic. For both tactics, a hyperlink to their respective details pages is given. In case the ontology contains information about which resources describe that relationship, this information will be listed in the `resources` JSON array, for example for the *Proxy* pattern in the relationship *"alternative to pattern"*.

In the `itemAbstract` property, optionally some additional information about the element is shown, for example for the *Broker Purpose* element. The content of this property can either be the Wikipedia abstract that would show up when opening the detail page of that element, or a quote from a resource that was added in the ontology. The Flask tool will select an appropriate text to show in the `itemAbstract` property. The purpose of this abstract is to provide the user with a quick overview of what this related item is about, without having to click on it. Of course, the user can click on it if the information in the abstract does not suffice.

In addition, if a view belongs to a viewtype, this is summarised in the *viewtypes* JSON array.

```
{
    "catalogUrl": "/catalog/pattern/BrokerPattern",
    "name": "Broker",
    "ontologyUrl": "http://thesis.wouterpinnoo.be/patterns.owl#BrokerPattern",
    "title": "Broker (Pattern)",
    "wiki": {
        "url": "http://en.wikipedia.org/wiki/Broker_Pattern",
        "abstract": "The Broker architectural pattern can be used to structure
            distributed software systems with decoupled components that interact by
            remote service invocations. A broker component is responsible for
            coordinating communication, such as forwarding requests, as well as for
            transmitting results and exceptions.",
        "resources": [
            {
                "googleBooksLink": "http://books.google.co.uk/books?
                    id=gJjgAAAAMAAJ\u0026dq=0471958697\u0026hl=\u0026source=gbs_api",
                "label": "Frank Buschmann, Regine Meunier, Hans Rohnert. Pattern
                    Oriented Software Architecture Volume 1: A system of patterns,
                    volume 1, 1996."
            },
            {
                "googleBooksLink": "http://books.google.co.uk/books?
                    id=mdiIu8Kk1WMC\u0026dq=0321154959\u0026hl=\u0026source=gbs_api",
                "label": "Len Bass, Paul Clements, and Rick Kazman. Software
                    Architecture in Practice, volume 2. 2003."
            }
        ],
        "relationships": {
            "has purpose": {
                "Broker": {
                    "itemAbstract": "By using the Broker pattern, an application can
                        access distributed services simply by sending message calls
                        to the appropriate object. instead of focusing on low-level
                        inter-process communication. In addition, the Broker
                        architecture is flexible, in that it allows dynamic change,
                        addition, deletion, and relocation of objects.",
                "name": "Broker",
                "resources": [

                ],
                "type": "Purpose",
                "url": "/catalog/purpose/BrokerImplPurpose"
            }
        },
        "has view": {
            "Broker": {
                "name": "Broker",
                "resources": [

                ],
                "type": "View",
                "url": "/catalog/view/BrokerImplView",
                "viewtypes": [
                    "Distribution",
                    "Component Connector"
                ]
            }
        },
        "has structure": {
            "Broker": {
                "itemAbstract": "The Broker architectural pattern comprises six
                    types of participating components: clients. servers,
                    brokers, bridges, client-side proxies and server-side
                    proxies.",
                "name": "Broker",
                "resources": [

                ],
                "type": "Structure",
                "url": "/catalog/structure/BrokerImplStructure"
            }
        },
        "alternative to pattern": {
            "Proxy": {
                "name": "Proxy",
                "resources": [
                    {
                        "resourceGoogleBooksURL": {
                            "googleBooksLink": "http://books.google.co.uk/books?
                                id=gJjgAAAAMAAJ\u0026dq=0471958697\u0026
                                hl=\u0026source=gbs_api",
                            "label": "Frank Buschmann, Regine Meunier, Hans
                                Rohnert. Pattern Oriented Software Architecture
                                Volume 1: A system of patterns, volume 1, 1996."
                        }
                    }
                ],
                "type": "Pattern",
                "url": "/catalog/pattern/ProxyPattern"
                },
                (...)
            },
            "comprises tactic": {
                "Use Intermediary": {
                    "name": "Use Intermediary",
                    "resources": [

                    ],
                    "type": "Tactic",
                    "url": "/catalog/tactic/UseIntermediaryTactic"
                },
                "Restrict Communication Paths": {
                    "name": "Restrict Communication Paths",
                    "resources": [

                    ],
                    "type": "Tactic",
                    "url": "/catalog/tactic/RestrictCommunicationPathsTactic"
                },
                (...)
            },
            (...)
        }
    }
}
```

Listing 6.6: Example JSON output for the details page of the Broker pattern

6.4 Conclusion

This chapter explained the technical decisions made in the design and implementation of the used components. The architecture of the proposed system was depicted and discussed in detail. Although the technical decisions and the architecture don't influence the ontology, it does influence the way users experience the work product of this master dissertation: the web-tool. For example, choosing the right SPARQL engine impacts the execution time of queries and as a consequence also the time it takes to load a page in the web-tool.

Chapter 7

Evaluation

In this chapter, an evaluation is made on the work product of this master dissertation (ontology and web-tool) with the purpose of self-reflection. The result is compared with goals set in the initial phase.

7.1 Ontology

The ontology developed in Chapter 3 forms the basis of this master dissertation. It was developed after an elaborate literature research and its components were chosen by looking at the most important literature resources and by finding out which components are necessary for the ontology for being able to include as much information as possible. The following sections state questions that are used to evaluate the result.

7.1.1 Which design modifications could improve the ontology?

The design of the ontology was, as described in Chapter 3, designed starting from the concepts defined in the ISO-42010 standard [22] and concepts found in literature that seem to be relevant for architects. Most of the theoretical descriptions of architectural patterns can be (partially) included in the ontology. This however does not mean that the design is perfect, nor that all components in the ontology are useful.

As first part of the self-reflection on the designed ontology, literature resources were again compared to the finished ontology and a conclusion was made that some components of the ontology were only applicable on architectures and patterns that were already implemented, instead of being able to capture theoretical information of patterns. A theoretical description describes a pattern in terms of how to implement it in a given context, while documentation of patterns that are already implemented contain more information on the interacting components of the pattern — the components are already established and can be easily documented in terms of which quality attributes they achieve, which purposes they have, etc.

This confusion probably arose when studying literature in which theoretical descriptions of patterns also contain elaborate examples of the components that are used in the pattern. These examples lead to the design of the ontology in which the *Structure* component of a pattern is formed by a set of *Components*, which in turn consist of a set of *Collaborators* and *Responsibilities*. These components are hard to define when adding a pattern that has yet to be applied in a concrete context. Although a pattern description can have a prescription of its internal structure, only an application of the pattern to a system will reveal details about the interacting components and their responsibilities in the structure of the pattern. As a conclusion, removing the *Component*, *Collaborator* and *Responsibility* components from the ontology would improve its applicability to theoretical descriptions of patterns. As a consequence, the *Structure* component will have to be used as a container for plain-text information about the structure of the pattern.

A second item of self-reflection on the design of the ontology is the definition of the components *Viewpoint*, *View* and *Viewtype*. The definition of these concepts in literature is quite abstract, and for someone trying to add a pattern the ontology, it might not be very clear which information should be added in which component. A distinction between these concepts was made because their definitions made clear that they should not be confused with each other. However, the amount of available information about views and viewpoints of patterns is rather limited, which questions whether all these concepts should be defined in the ontology. Probably only one of them, e.g. viewpoints, would suffice, and information could be added in plain-text in that component, so that text-based (possibly vague) descriptions of view(points) can be included in the ontology.

In addition, to avoid that the ontology is a representation of information of patterns that are already implemented, all information contained in the *View* components can be moved to the *Viewpoint* components. The *View* components can then be removed from the ontology — since *Views* are defined as implementations of a *Viewpoint* [22, 10].

7.1.2 Is the ontology generic enough?

Although the information found in these resources was successfully included in the ontology, the ontology must also be able to include information from new sources. Therefore in this section the ontology is evaluated on whether it is capable to cope with this.

Buschmann et al. [7] describe five concepts that are used in pattern languages and that should be considered when designing a software system. These concepts are explained below, along with a description on how the ontology is compatible with them.

- **Pattern complements:** complements are sets of patterns that are often used along with each other. These patterns mostly share a common problem family. The ontology is compatible with this kind of relationship with the `isCombinableWith` property. The authors suggest that other relationships can also be defined, for example *"pattern A complements pattern B"* or *"pattern A cannot be used along with pattern B"*. This is not included in the ontology because it was noticed that these relationships are seldom used in practice — or at least seldom used in literature.
- **Pattern compounds:** these are in other words groups of patterns. Patterns can be grouped if they form a recurring cohesive combination and thus form a new pattern. This is, in a simplified way, supported in the ontology with the `uses` and `isUsedIn` properties.
- **Pattern sequences:** sequences are stories of how to apply patterns in a given situation. These stories can be very helpful for an architect but are a question of good documentation rather than implementation in the ontology. Stories don't fit very well in this ontology because the ontology concentrates on patterns and not on the process of applying them. However it is probably a good idea to develop a second ontology, similar to this one, that implements pattern sequences and uses the *Pattern* class (and its instances) from the current

ontology. As the authors describe, in expressing an architecture an important difference has to be made between spatial relationships and temporal relationships. The former are those of connection by inclusion and consists of those used in the ontology. The latter are those of sequencing in time, which is what pattern sequences implement. Those two types of relationships can be implemented in two different ontologies.

- **Pattern collections:** this is the most important one in this master dissertation since it focusses on organising patterns based on common properties. The organisation types in pattern collections are:
  - by level: a distinction between design patterns and architectural patterns. Only architectural patterns are considered in this master dissertation.
  - by domain: the two main domains are the problem domain and the solution domain. Patterns in the problem domain are suited for situations like the decomposition of problems, while those in the solution domain focus on situations in the resulting software: the internal structure, the used technologies,. . . The *Environment* component of the ontology supports making this distinction, for example by implementing two subclasses of the *Environment* class (*ProblemDomain* and *SolutionDomain*) and using instances of these classes to define the domain more specifically.
  - by partition: a distinction is made here by which part of the architecture is involved when the pattern is applied. For example, patterns can be grouped by those that are used in the presentation tier and those in the business tier. The ontology does not contain a component that is explicitly designed for this classification but the above mentioned *Environment* component can be slightly modified such that it can also be used for classification by partition, by adding child components to it.
  - by intent: the authors conclude that there are different kinds of intents. The ontology supports the most important ones: structures, purposes and the tactics that are achieved by the pattern.

These concepts seem to fit quite well in the current ontology, only little modifications are needed to the ontology to support them all.

Another point of view of checking whether the current ontology is generic — whether it is capable to capture information from new sources — is looking at the templates that are used in literature to document patterns. Literature resources that list patterns and document them use mostly a fixed way of structuring the information. In an ideal case, each section can be mapped to a component in the ontology, such that all information can be comprised in the ontology. Some templates and their compatibility with the ontology are discussed below.

- **Bass et al.** [5]: pattern documentations in this book are relatively short and mostly plain text. First, a context of the pattern is given. The information in this text part is more an abstract of the pattern, rather than a strict description of the context in which the pattern should be used. The ontology can contain this information by simple adding a `String` annotation to the pattern object. Next, the problem and solution are briefly described. The former can be mapped to purposes and tactics, while information from the latter can be mapped to the *Structure* component. Also, the solution description contains references to other patterns, e.g. with the "`uses`" relationship.
  Finally, the template contains a table that describes following items:
  - *Overview:* an abstract of the pattern. This can be included in the ontology with annotations.
  - *Elements:* a list of all components in the pattern along with brief descriptions. Elements of an implementation of the pattern would fit in the ontology with the *Structure* component and a set of *Collaborators* with their *Responsibilities*. However, as described in Section 7.1.1, these components are only suited for pattern implementation documentation (not for general pattern documentation) and can be removed from the ontology.
  - *Relations:* a list of relations between elements. Again, such relationships only occur between elements of actual pattern implementation and thus don't fit in this ontology.
  - *Constraints:* a list of restrictions on the components of the pattern. Constraints such as *"this pattern should not be used in the environment Y or when trying to achieve quality attribute X"* could be added to the ontology by, for example, defining a relationship `isNotCompatibleWith`. However, most of the constraints found in this section of the template are

constraints on the actual implementation of the pattern (such as *"element X cannot be used more than once"*), which don't fit in the ontology.

  - *Weaknesses:* these can be included in the ontology by adding them as *Consequence* objects. In case they can be related to a tactic, the ontology supports adding this link too.

- **Bushmann et al. [8]:** pattern documentations are in this book more extensive. The template starts with a short abstract of the pattern. Again, this can be added in the ontology with an annotation. Next, the following items are described:

  - *Example:* a text-based example is given. This fits in the *Example* component of the ontology. The text of the example can be added with an annotation on the corresponding *Example* object.
  - *Context:* a short description of the targeted context. In this book, the context is actually a list of purposes of the pattern, so this fits in the *Purpose* component of the ontology, on the condition that this component is sufficiently defined with a clear (plain-text) description of the *purpose*.
  - *Problem:* this is a larger, more extensive description of the problem in the given context. The information it contains can be summarised in *Purposes*.
  - *Structure:* the elements of the pattern are expressed in Class-Responsibility-Collaborator (CRC) cards, which fits perfectly in the corresponding components of the ontology. However, this section also contains all interactions between the components with detailed examples. This is harder to include in the ontology, but considering the level of detail this should not be included at all in component-form since it is not useful in categorisation of patterns. Thus, the ontology can simply contain a link to the information in the book (or alternatively include the plain text).
  - *Implementation:* a guide on how to use this pattern. Again, a link to this information can be added to the ontology instead of trying to map it to ontology components.
  - *Variants:* these are implemented in the ontology with inheritance of properties of patterns. For example, the *Message Passing Broker System* pattern is a variant of the *Broker* pattern because its view is an extension of

the view of the *Broker* pattern. Those patterns are linked in the ontology by inheriting the view of the first one from the view of the second one.

- *Known uses:* can be implemented with the `uses` property.
- *Consequences:* this fits the *Consequence* component in the ontology.
- *See also:* this section lists patterns with yet other relationships to the current pattern. Most relationships are already present in the ontology (e.g. `isUsedIn`, `isAlternativeTo`,...).
- *Credits:* this section contains optional references to other resources, which fits the resource annotation method as discussed in Section 3.3.

• **Fowler [15]:** pattern descriptions from this literature resource were not used at all to construct the ontology, so this is certainly a good source to use in this evaluation. In addition, it contains both design patterns and architectural patterns. The template of this resource starts, like most templates, with an abstract that can be added as annotation. The abstract sometimes also contains information that can be added in the ontology as *Purpose* objects. The other sections in the template are the following:

  - *How it Works:* this section lists all elements and explains how the elements work. The description is focused on the practical implementation of the patterns rather than their structure. Therefore this information is hard to include in the ontology, since the ontology is designed to document the theory of the patterns rather than their implementations (see Section 7.1.1).
  - *When to Use It:* again, the description in this section is quite practical, but several *Purpose* and *Tactic* components can be identified from the text. The practical descriptions can eventually be included as annotations.
  - *Example:* a detailed example with code examples (for the design patterns). This fits in the *Example* component in the ontology.
  - *Further reading:* contains references to literature resources, which can be added in the ontology as described in Section 3.3.

  It was noted that the template in this resource contains less explicit elements while more information is added as plain text. There is less structure to be found in the template. This makes it harder to include information in the ontology, since extracting the useful information will be time-consuming.

As a conclusion, it can be stated that the ontology can not contain *all* information found in literature resources, but certainly enough to use it for categorisation purposes. Since some literature resources have pattern descriptions in plain text, most of the information will be contained in ontology annotations. Also, a lot of the information found in literature is bound to the interpretation of the author. The purpose of the ontology is to only contain unambiguous information, unless data is annotated with a user reference.

7.2 Web-tool

An evaluation of the web-tool was made by observing five users using the tool for a given set of use cases.

First, the manual from Appendix A was given to the users. There was no test performed on the intuitiveness of the tool since the focus of this master dissertation was not to design the most optimal user interface but rather the underlying functionality, and the question of developing a system where architects have all the functionality they need in searching patterns. Therefore users in this test had to fully read the manual so that they have full knowledge of the available functionality, and on which pages they can find it.

Next, the users had to solve questions or perform use cases. They are chosen such that they represent the actions architects will most likely perform on this tool. The actions of the users were observed. They were also asked to explain all actions they took.

7.2.1 Use cases, questions and results

The questions are listed below, along with a description of the actions of the users and a conclusion.

1. **Use case:**
   Design a system for an emergency call-centre: the call takers must be able to

continuously input information. The dispatcher receives this information from multiple call takers (for the same incident) and highlights some information that has to be passed on to the emergency services (some information to the fire department, other information to the police department).

- **Question:**
  What are relevant architectural patterns?
  Results:
  It was clear to all users that this has to be found on the home page of the tool. All users made clear that the categorisation method *Quality attributes and corresponding tactics* and *Structures* were not relevant in this case. However, it was for some users unclear whether they had to search for a *Purpose* or a *Viewtype*, although I expected the users to search for the viewtype *Dataflow*.

- **Question:**
  Which tactics (and therefore also which quality attributes) are addressed by patterns that are relevant in this situation?
  Results:
  As expected, the users searched for tactics by clicking on a pattern from the right column on the home page. If the pattern comprises a tactic, this will be mentioned on the page that contains the details of the pattern.

2. **Question:**
   Are there any patterns addressing interoperability that are used in dataflow systems?
   Results:
   Since in the previous question users had to click on a pattern in the right column to view more information about it, I expected that users would tend to use this method again which only two out of five did. The other users recalled that the manual states that a filter on patterns can be defined by selecting items from different categorisation methods. In this case, the items *Interoperability* from the categorisation method *Quality attributes* and *Dataflow* from *Viewtypes* had to be selected.

3. **Question:**
   I have a dataflow system. List all patterns that could be relevant in my situation

(directly or indirectly).
Results:
All of the users simply selected *Dataflow* in the left column and retrieved the list of patterns. However, the hint *"directly or indirectly"* was meant to recall that there is a relationship defined in the ontology that tells us which patterns are combinable with a given pattern. Those patterns can be retrieved from the home page by clicking on a pattern in the right column. Unfortunately, none of the users had knowledge of this and thus did not search for this relationship. This implies that the manual should be extended with a section that describes in a non-technical way the relationships that are defined in the ontology. Alternatively, the tool could be extended with a list of all relationships in the ontology, which will make it more future proof: if a relationship is added to the ontology, the manual does not have to be updated.

4. **Question:**
I'm trying to make my system highly modifiable. Which patterns should I use and which literature resources should I consult for this?
Results:
All users succeeded in finding these resources: using the home page to find relevant patterns and then clicking on them to view their corresponding literature resources. One user however first used the catalog to find the resources. He clicked on the *Quality attributes* tab, selected *Modifiability* and expected that all resources of all patterns related to modifiability were included on that page. He made a good point to do this. The page would however contain too much irrelevant information since it will contain also literature resources of patterns in which only the patterns are described and not the relation to tactics that achieve modifiability.

5. **Question:**
I'm not sure what a Broker pattern does. Give a one-liner that describes this pattern.
Results:
Presumably because the users have had earlier questions about resources, they all knew that some patterns contain a Wikipedia abstract at the top of the page. They all used the catalog page to find the Broker pattern directly. The manual also contained a reference to this.

6. **Question:**
   List all tactics that achieve modifiability. I'm not interested in inheritance of tactics — I consider *tactics* and *child-tactics* as the same.
   Results:
   Four out the five users were looking at the home page, and simply expanded the tree in the left column to see all tactics that can be found under the item *Modifiability*. The same user that opened the detailed page of *Modifiability* in question 4, navigated again to this page and saw three tactics listed on that page. He was unaware that these tactics had child-tactics. This suggests that the tool could be upgraded to indicate whether an item on a detail page has children, without having to click on that item.

7. **Question:**
   Add a pattern *TestPattern* that is a variant of the *Broker* pattern — it has the same structure but adds some components.
   Results:
   Surprisingly, the users had some trouble finding how to do this action, although it was described clearly in the manual. Three users were searching on the home page where to add an item and then immediately looked in the manual and followed the exact instructions. They recalled that there was an image showing the *Add new pattern* button, and searched for this image to find the instructions. Perhaps this is not the right way to search in a manual: the manual could be extended with a table of contents.
   The other two users were searching on the catalog page for the button and didn't find it because the button only appears when the Stardog engine was selected, which was not the case. These users both gave the feedback that the button has to be visible in any case, and that it should be disabled when the Stardog engine is not selected.

8. **Question:**
   The goal of my system is certainly not to be distributed. Which patterns are relevant that comprise the *Encapsulate* tactic?
   Results:
   Four out of five users solved this question the same way that was intended: using the browser page. They mentioned that they used this because it seemed easier to use than the homepage in this use case. They expanded the tree to

all relevant patterns and manually filtered those patterns that comprise the *Encapsulate* tactic and not have the *Distributed System* purpose. One user used the home page: she selected the *Encapsulate* tactic and all items in the *Purpose* section except the *Distributed System* purpose. This yields the same results, but the total time needed was significantly larger because the constructed query was much longer (the query included all sets of patterns that had to be excluded from the query response.).

9. **Question:**
   Look for resources that describe the connection between the *Publisher-Subscriber* pattern and the *Component-Connector* viewtype.
   Results:
   The four users looking at the browser page (because they used it in the previous question) navigated in the tree to the *Component-Connector* viewtype but were stuck at that point. They realised that the browser page can not be used to navigate to the details page of a viewtype. Two users used the catalog to navigate to the *Component-Connector* viewtype page which has a direct link to the *Publisher-Subscriber View* page. That page contains the required information. Both the other two users and the fifth user used the catalog to navigate to the *Publisher-Subscriber Pattern* page and then navigated to the *Publisher-Subscriber View* page.

7.2.2 General remarks

It was noted that many users didn't have the patience to wait for a query to end on the homepage after clicking the *Look up patterns* button. Although the cursor is changed to indicate that the page is loading, users tend to start another query hoping it will be faster. One user gave the suggestion to immediately execute the query (without pressing the button) because it is more intuitive to use a live filter. However this is practically not usable because the delay on the live filter would be too big.

At a random moment during the tests, I disabled server-side caching to observe the impact. When a new page loaded, some users tended to close the page before it was done loading, and instead opened the browser page. For example, for one user I disabled server-side caching after question 6, and requested to redo his actions.

After he noticed that the loading time of the page that contains details about the *Modifiability* quality attribute increased significantly, he switched to the browser page to view all relevant tactics. This indicates that this page is indeed more suitable for the purpose of quickly listing children or relationships of an item.

Chapter 8

Conclusion and future work

8.1 Conclusion

This master dissertation focused on tackling the problem of software architects in not finding enough relevant information on architectural patterns, and spending too much time in researching literature for relevant information.

Existing literature on this subject was researched with attention to existing classification methods of architectural patterns. Optimising a classification of architectural patterns can help defining a way to both suggesting related information and clustering patterns such that patterns can be found given the cluster details.

A classification schema was designed in Chapter 2 that generalises existing classification methods in combination with a better definition of used concepts about pattern documentation in literature. This classification schema gave insights in which components of an architectural pattern are most relevant in the purpose of making a classification schema that will be applicable in a more general sense.

The gained insights from the classification schema formed the basis of making an ontology for architectural patterns and their related concepts, as proposed in Chapter 3. This ontology is made with attention to modifiability and extensibility. It hopefully can serve as a basis for future work to define even more concepts in this ontology. The ontology was designed to be able to include all information from the classification schema in the ontology. For example, one of the classification methods in the schema is based on *Viewtypes*. The ontology will define all *Viewtypes* and all relationships with patterns and other concepts. The ontology also annotates each concept, instance and relationship with resources. This way, the ontology will not only be useful for retrieving information but also for linking resources to information.

A querying manual on this ontology and a visualisation for querying was designed in Chapter 4 and 5 respectively. Architects can with this tool easily retrieve a list of architectural patterns after choosing items from a category list. For example, an architect can choose to retrieve all patterns related to tactics that achieve a certain quality attribute, while constraining the results to only those patterns with a certain internal structure. In addition, the tool can provide the architect with all information that is related to a certain object that was found in the ontology. This information will also include in detail which resources are defining the information, so that the architect can rely on external resources while using this tool as a guiding system in finding where to look to relevant information resources.

Finally, an evaluation and self-reflection of both the ontology and the web-tool were made in Chapter 7.

8.2 Future work

As a first improvement suggestion, as discussed in Section 7.1.1, the ontology can be modified to include only theoretical descriptions of patterns, along with text-based annotations; rather than trying to fit each and every aspect of a pattern description in an ontology component.

Next, in terms of resource linking, the ontology can be extended to include more types of resources. The types of resources currently supported is limited to URLs, ISBN numbers, plain text and links to the DBpedia ontology. A suggested future extension is supporting PDF-files and HTML documents. These documents can then with little modification be included in the web-tool.

The current web tool is designed to support all kinds of pattern retrievals, but could be more extended to support adding objects to the ontology. A prototype of this was given in Chapter 5 but doesn't support all relationships of the ontology. More elaborate research is possible in this domain: more specifically which visualisation methods would be best suitable for adding annotations to relationships, adding relationships in the ontology, etc., while keeping the ontology consistent. No modifications are needed on the ontology itself for this.

The web tool also supports server-side caching of the catalog pages. A cache implementation of these pages was made in the prototype to decrease loading times during testing. However, for practical usage of the web tool in production, the cache can be moved to the client side, e.g. in the browser cache. This will also make it possible to define the preferences for each user: when one user changes its preferred SPARQL engine, it won't affect the used engine for other users.

Lastly, as discussed in Section 4.4, a live reasoner on the SPARQL engine can be beneficial in terms of capability of data manipulations without harming the consistency of the data. Inferred data was generated in the current version and included in the original ontology. Instead, the original ontology can be loaded in an engine supporting live reasoning.

Bibliography

[1] Akerman, A. and Tyree, J. *Using ontology to support development of software architectures*. IBM Systems Journal, 45(4):813–825, 2006. ISSN 0018-8670. doi: 10.1147/sj.454.0813.

[2] Apache. *Apache Jena - Fuseki: serving RDF data over HTTP*. URL `https://jena.apache.org/documentation/serving_data/`. Last accessed: 16-03-2016.

[3] Auer, S., Dietzold, S., and Riechert, T. *OntoWiki - A Tool for Social , Semantic Collaboration*. Science, 4273:736–749, 2006. ISSN 03029743. doi: 10.1007/11926078_53.

[4] Avgeriou, P. and Zdun, U. *Architectural Patterns Revisited - A Pattern Language*. Information Systems Journal, 81:1–39, 2005. doi: 10.1.1.141.7444.

[5] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, volume 2. 2003. ISBN 0321154959.

[6] Bizer, C., Cyganiak, R., Auer, S., and Kobilarov, G. *DBpedia.org—Querying Wikipedia like a Database*. In Developers track at 16th International World Wide Web Conference (WWW2007), Banff, May 2007.

[7] Buschmann, F., Henney, K., and Schmidt, D. C. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, volume 5. Wiley Publishing, 2007. ISBN 0471486485.

[8] Bushmann, F., Meunier, R., and Rohnert, H. *Pattern-Oriented Software Architecture Volume 1: A system of patterns*, volume 1. Wiley Publishing, 1996. ISBN 0471958697. doi: 10.1192/bjp.108.452.101.

[9] Choobdaran, N., Sharfi, S. M., and Khayyambashi, M. R. *An Ontology-Based Approach For Software Architectural Knowledge Management*. Journal of Mathematics and Computer Science, 11:93–104, 2014.

[10] Clements, P., Garlan, D., Little, R., Nord, R., and Stafford, J. *Documenting software architectures: views and beyond*. 25th International Conference on Software Engineering, 2003. Proceedings., pages 3–4, 2003. ISSN 0270-5257. doi: 10.1109/ICSE.2003.1201264.

[11] Complexible Inc. *Stardog: Enterprise Data Unification with Smart Graphs*. URL `http://stardog.com/`. Last accessed: 15-05-2016.

[12] de Graaf, K. A., Tang, A., Liang, P., and van Vliet, H. *Ontology-based Software Architecture Documentation*. 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, pages 121–130, 2012. doi: 10.1109/WICSA-ECSA.212.20.

[13] Dietrich, J. and Elgar, C. *A formal description of design patterns using OWL*. Software Engineering Conference, 2005. Proceedings. 2005 Australian, pages 243–250, 2005. ISSN 1530-0803. doi: 10.1109/ASWEC.2005.6.

[14] Flask. *Flask: A Python Microframework*. URL `http://flask.pocoo.org/`. Last accessed: 16-03-2016.

[15] Fowler, M. *Patterns of Enterprise Application Architecture*, volume 23. 2003. ISBN 0-321-12742-0. doi: 10.1119/1.1969597.

[16] Fowler, M. and Highsmith, J. *The agile manifesto*. Software Development, 9 (August):28–35, 2001. ISSN 10708588. doi: 10.1177/004057368303900411.

[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

[18] Gruber, T. R. *A Translation Approach to Portable Ontology Specifications*. Knowledge Creation Diffusion Utilization, 5(April):199–220, 1993. ISSN 1042-8143. doi: http://dx.doi.org/10.1006/knac.1993.1008.

[19] Henninger, S. and Ashokkumar, P. *An ontology-based metamodel for software patterns*. 18th International Conference on Software Engineering and Knowledge Engineering, SEKE 2006, pages 327–330, 2006.

[20] Horrocks, I. *Description Logic : A Formal Foundation for Ontology Languages and Tools*. Methods In Cell Biology, 78(0091-679X (Print) LA - eng PT - Journal Article RN - 0 (Intermediate Filament Proteins) RN - 0 (Nerve Tissue Proteins) RN - 0 (Sulfur Radioisotopes) RN - 63-68-3 (Methionine) SB - IM):765–775, 2007.

[21] IEEE-SA Standards Board. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Std, 1471-2000:1–23, 2000. ISSN 1471-2000. doi: 10.1109/IEEESTD.2000.91944.

[22] International Organization Of Standardization. *ISO/IEC/IEEE 42010:2011 - Systems and software engineering – Architecture description*. ISOIECIEEE 420102011E Revision of ISOIEC 420102007 and IEEE Std 14712000, 2011 (March):1–46, 2011. doi: 10.1109/IEEESTD.2011.6129467.

[23] Jansen, A., Avgeriou, P., and van der Ven, J. S. *Enriching software architecture documentation*. Journal of Systems and Software, 82(8):1232–1248, Aug. 2009. ISSN 01641212. doi: 10.1016/j.jss.2009.04.052.

[24] Kalibatiene, D. and Vasilecas, O. *Survey on ontology languages*. In Lecture Notes in Business Information Processing, volume 90 LNBIP, pages 124–141, 2011. ISBN 9783642245107. doi: 10.1007/978-3-642-24511-4_10.

[25] Kampffmeyer, H. *Formalization of Design Patterns by Means of Ontologies*. GRIN Verlag, 2007. ISBN 3638615081.

[26] Maplesden, D., Hosking, J., and Grundy, J. *Design Pattern Modelling and Instantiation using DPML*. Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications - CRPIT '02, 10:3–11, 2002.

[27] Microsoft Patterns & Practices Team. *Microsoft Application Architecture Guide*. Microsoft Press, 2009. ISBN 9780735627109.

[28] Mikkonen, T. *Formalizing Design Patterns*. In Proceedings of the 20th International Conference on Software Engineering, pages 115–124, Kyoto, Japan, 1998. IEEE Computer Society.

[29] Miles, J. *Bootstrap Treeview*. URL `http://jonmiles.github.io/bootstrap-treeview/`. Last accessed: 04-05-2016.

[30] Nord, R. L. and Tomayko, J. E. *Software architecture-centric methods and agile development*. IEEE Software, 23(2):47–53, 2006. ISSN 07407459. doi: 10.1109/MS.2006.54.

[31] Opdahl, A. L., Berio, G., Harzallah, M., and Matulevičius, R. *An ontology for enterprise and information systems modelling*. Applied Ontology, 7(1):49–92, 2012. ISSN 1570-5838. doi: 10.3233/AO-2011-0101.

[32] Osterwalder, A. and Pigneur, Y. *Clarifying Business Models : Origins , Present , and Future of the Concept*. Communications of Association for Information Systems, 15(May), 2005.

[33] Pahl, C., Giesecke, S., and Hasselbring, W. *Ontology-based modelling of architectural styles*. Information and Software Technology, 51(12):1739–1749, 2009. ISSN 09505849. doi: 10.1016/j.infsof.2009.06.001.

[34] Rozanski, N. and Woods, E. *Using Architectural Perspectives*. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pages 25–35, 2005. doi: 10.1109/WICSA.2005.74.

[35] Rozanski, N. and Woods, E. *Applying Viewpoints and Views to Software Architecture*. Computer methods and programs in biomedicine, (2005):11, 2011. ISSN 1872-7565. doi: 10.1016/j.cmpb.2011.06.008.

[36] Rozanski, N. and Woods, E. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2011. ISBN 0132906120.

[37] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, volume 123. 1996. ISBN 0131829572.

[38] Stanford Center for Biomedical Informatics Research. *ProtegeWiki: SQWRL*. URL `http://protege.cim3.net/cgi-bin/wiki.pl?SQWRL`. Last accessed: 25-04-2016.

[39] Stanford Center for Biomedical Informatics Research. *Protégé*. URL `http://protege.stanford.edu/`. Last accessed: 15-05-2016.

[40] Suter, R. *Software Design and Architectural Patterns*. URL `http://software-pattern.org/`. Last accessed: 28-05-2016.

[41] Taibi, T. and Ngo, D. C. L. *Formal Specification of Design Patterns - A Balanced Approach*. Journal of Object Technology, 2(4):127–140, 2003.

[42] Tang, A. and Van Vliet, H. *Building Roadmaps : A Knowledge Sharing Perspective*. Technology, pages 13–20, 2011. ISSN 02705257. doi: 10.1145/1988676.1988681.

[43] Tang, A., Liang, P., and Van Vliet, H. *Software Architecture Documentation: The Road Ahead*. 2011 Ninth Working IEEE/IFIP Conference on Software Architecture, pages 252–255, 2011. doi: 10.1109/WICSA.2011.40.

[44] W3C. *Frequently Asked Questions on W3C's Web Ontology Language (OWL)*. URL `http://www.w3.org/2003/08/owlfaq`. Last accessed: 25-04-2016.

[45] W3C. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. URL `https://www.w3.org/Submission/SWRL/`. Last accessed: 25-04-2016.

[46] W3C. *SPARQL: Query Language for RDF*. URL `https://www.w3.org/TR/rdf-sparql-query/`. Last accessed: 15-05-2016.

[47] W3C. *RDF - Semantic Web Standards*. URL `https://www.w3.org/RDF/`. Last accessed: 15-05-2016.

Appendix A

Web-tool manual

In this manual, a description is given of how to use the tool that can retrieve architectural patterns. Table A.1 lists the pages in this tool, along with a brief description. Sections A.1 to A.4 describe the pages in detail. The pages can be retrieved with the navigation bar at the top, as depicted in Figure A.1, except for the *Item details* page since it is only accessible by clicking on an object in the *Catalog* page. In Section A.5, some additional features of the tool are explained.

Page	**Description**
Home page	This page is designed to retrieve architectural patterns, based on user input of categorisation methods, such as quality attributes, purposes, a.o.
Catalog	The catalog provides the user with a searchable list of all objects, grouped by type (patterns, tactics,...).
Item details	A Wiki-like page that contains all information about a selected object (a pattern, tactic,...).
Browser	A visualisation of relationships between all objects, used for browsing through objects and their inter-relationships.

Table A.1: Pages in the web-tool

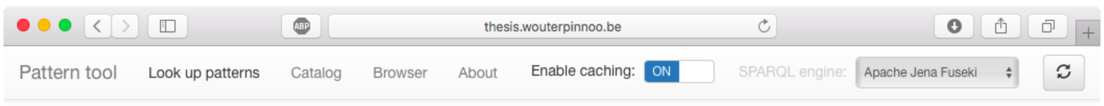


Figure A.1: Navigation bar

A.1 Home page

The home page of the tool (shown in Figure A.2) allows to retrieve a list of relevant architectural patterns based on categorisation information. In the left column, some criteria are listed that can be used to look up patterns. This is a hierarchical collapsible tree. Click on the small arrows to expand or collapse an item in the tree. The four grey items are main categorisation methods. Children of these items are categories, with subcategories as grandchildren. The main categorisation methods are:

- **Quality attributes and corresponding tactics**: the children are quality attributes that act as categories. Their children are all tactics that achieve the corresponding quality attribute. Tactics can have further hierarchical parent-children relationships.
- **Purposes**: these are general goals that patterns can achieve.
- **Viewtypes**: a viewtype is a category of those views that share a common set of elements and relationships. Children of this item are views, which are ways to represent a system.
- **Structures**: each child of this categorisation method is a category of those structures that share a common internal organisation of the pattern, i.e. have similar components.

A category can be selected in the left column by clicking on an item. The background of this item will change to blue. Deselecting can be done by clicking again on the item. If the selected item has children, all children will be selected too. Multiple items can be selected, even from different categorisation methods.

The search functionality can be used to find items in the hierarchical tree without navigating to it. Figure A.3 depicts the use of this functionality. When typing a keyword, a list with relevant items will be shown, along with the categorisation method they belong to. When an item from this list is selected, the corresponding item will be selected in the hierarchical tree.

By selecting multiple items in the hierarchical tree, filters can be defined for searching architectural patterns. For example, one can choose to select a tactic and a structure, which will limit the queried patterns to only those that comprise

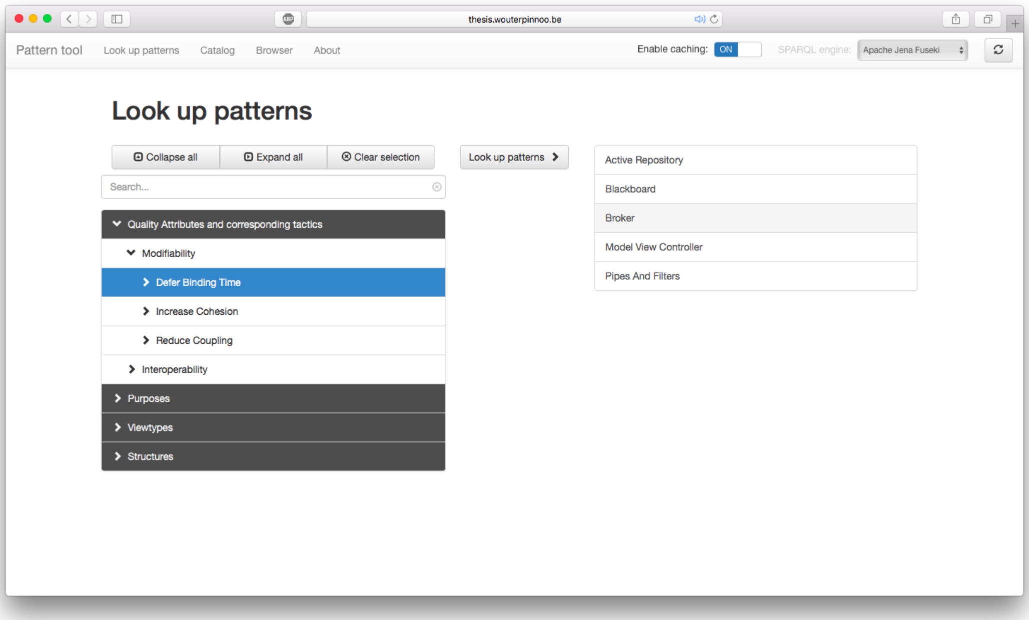


Figure A.2: Homepage

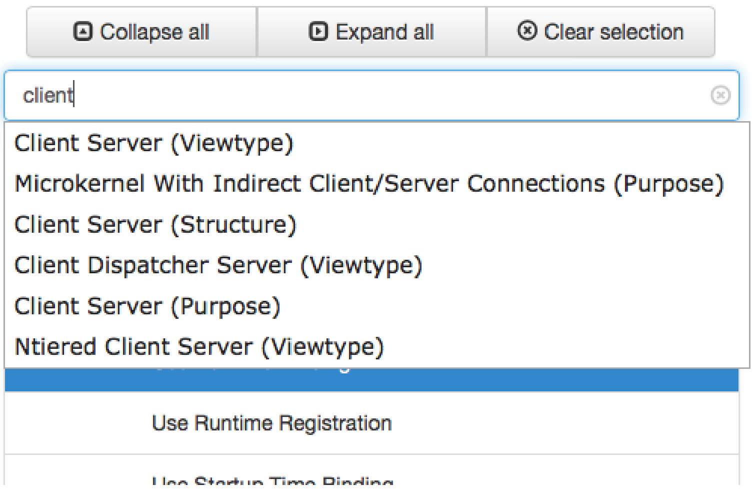
that tactic and inherit from the chosen internal structure. When two items from the same categorisation method are selected, the resulting list of patterns will be a concatenation of those patterns related to the first item and those related to the second item.

When the preferred selection is made, the architectural patterns can be queried by clicking on the *"Look up patterns"* button. After (at most) a couple of seconds, a list of relevant architectural patterns will appear in the right column. The details of a pattern can be viewed by clicking on the pattern, which will redirect the user to the *"Item detail"* page as will be discussed in Section A.3.

A.2 Catalog page

A screenshot of the catalog page is shown in Figure A.4. The type of object can be selected by clicking on one of the tabs at the top of the page. The corresponding list will be modified. The details of one item in the list can be viewed by clicking on it, which will redirect to the *"Item detail"* page (Section A.3).

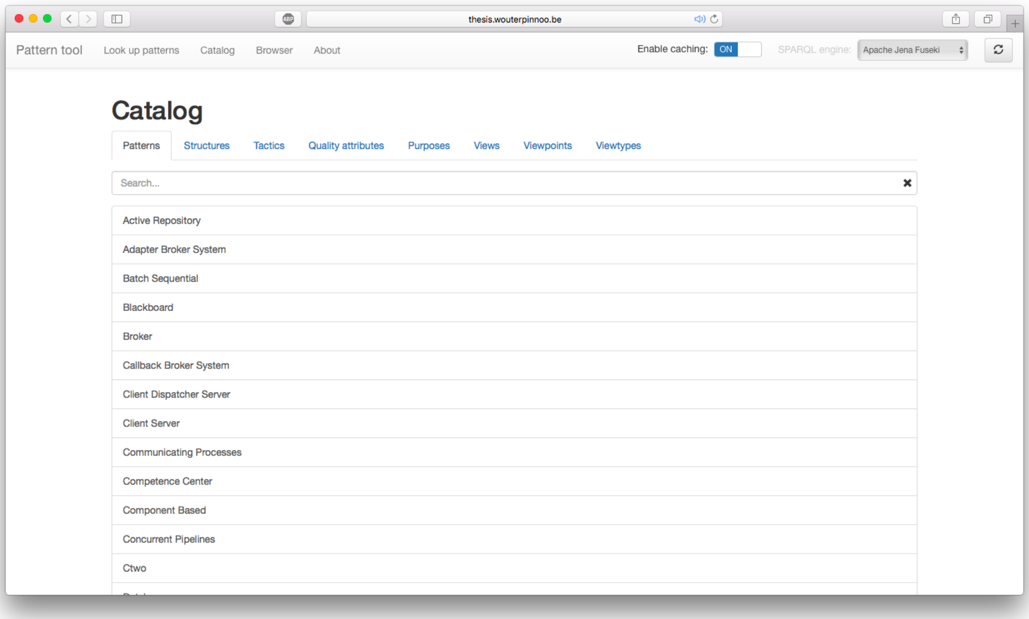
In case the Stardog SPARQL engine is selected (see Section A.5), a button will



Client Server (Viewtype)
Microkernel With Indirect Client/Server Connections (Purpose)
Client Server (Structure)
Client Dispatcher Server (Viewtype)
Client Server (Purpose)
Ntiered Client Server (Viewtype)

Figure A.3: Searching in the list of categorisation types

be visible to add items to the list, as shown in Figure A.5a. When clicking on this button, a similar dialog to that of Figure A.5b will appear. The contents of this dialog depend on which object type was selected on the top of the page (pattern, structure,...). In case the *pattern* object type was selected, the dialog provides input fields where relationships to tactics, purposes, a.o. can be defined. After filling in all required information, the object will be added to the system after clicking the *"Add"* button. The page will refresh after adding the item.



Enable caching: ON

SPARQL engine: Apache Jena Fuseki

Catalog

Active Repository
Adapter Broker System
Batch Sequential
Blackboard
Broker
Callback Broker System
Client Dispatcher Server
Client Server
Communicating Processes
Competence Center
Component Based
Concurrent Pipelines
Ctwo

Figure A.4: Catalog list

A.3 Item details page

This page contains all information known to the system related to the queried item. This page can be requested from either the home page (when clicking on a pattern in the right column) or from the catalog page (by clicking on an item in the list).

The shown content depends on which information is available in the system. If the system contains a link to a Wikipedia page, the abstract of the Wikipedia page is shown at the top of the page along with a link to the actual Wikipedia page.

Next, if applicable, all literature resources are listed that have somehow a connection with the requested item. If the system also contains an ISBN number of this resource, a link will be provided to the corresponding Google Books page.

Finally, all relevant relationships with this item are listed. For example, in Figure A.6 the requested object was the *Broker* pattern and the system seems to have knowledge of two relationships with this pattern:

1. the pattern has a structure, called *Broker*. Since this structure has the same name as the requested item, this is simply an object that contains all known details of the structure of the *Broker* pattern. Clicking on this item will redirect to the details page of this object.
2. the pattern comprises two tactics: the *Use Intermediary* tactic and the *Restrict Communication Paths* tactic. Again, click on those items to view the details of the tactics.

In some cases, the system has additional information of literature resources for such relationships. For example, the system can know in which book the relationship between the *Broker* pattern and the *Use Intermediary* tactic is described and on which page. If this information is known, it will show up next to the tactic name.

A.4 Browser

The browser page provides a more user-friendly tree-like visualisation of related relationships. The tree will initially show only the categorisation methods also listed on the home page. When clicking on one item, the tree will expand to show also all children of those items. When clicking on *Purposes* for example, all categories of *purposes* will be shown as child nodes.

When reaching a node that had no child nodes on the home page, clicking on this node will query all information known to this item. This is the same information that can be viewed on the *Item details* page. Nodes that show up as child nodes of a *purpose* node are for example *"belongs to pattern"* and *"addressed by view"*. These are the relevant relationships to the *purpose* object. The objects that are involved in that relationship can be queried by clicking on the corresponding node, and will cause the tree to expand further.

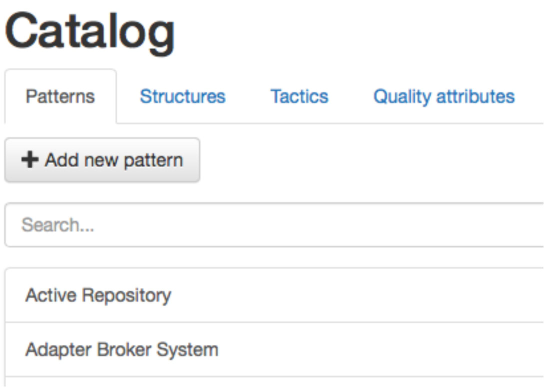
If the tree gets too big for your screen, simply click and drag the tree to navigate, and scroll to zoom in or out.

A.5 Advanced features

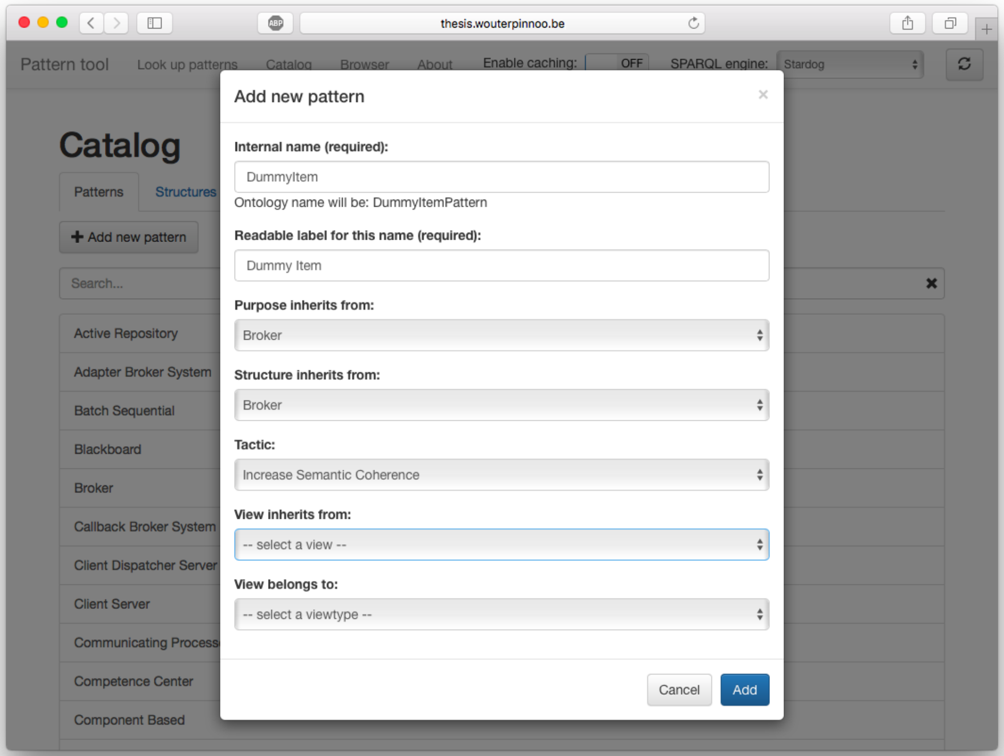
The tool allows to configure two user preferences:

Server-side caching With every page load, the system executes queries to retrieve relevant information. Because of this, it can sometimes take quite a long time to load the page. When server-side caching is enabled (see Figure A.7a), the server will store and reuse queried information.

Selection of SPARQL engine SPARQL engines are server components that query information. Several are implemented and — if the user is familiar with those engines — can be chosen in a dropdown list, as shown in Figure A.7b. After selecting a SPARQL engine, the server will remove all stored information in the caches and reload the page. Note that this option is disabled if caching is enabled.



(a) Button for adding items — only active when using the Stardog engine



Enable caching: OFF

SPARQL engine: Stardog

Add new pattern

Internal name (required): DummyItem

Ontology name will be: DummyItemPattern

Readable label for this name (required): Dummy Item

Purpose inherits from: Broker

Structure inherits from: Broker

Tactic: Increase Semantic Coherence

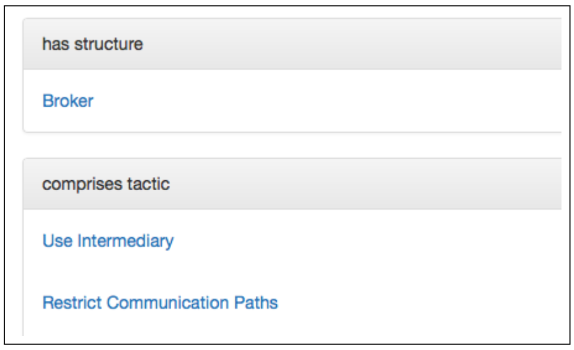
View inherits from: -- select a view --

View belongs to: -- select a viewtype --

Cancel Add

(b) Dialog for adding objects to the ontology

Figure A.5: Add item to the catalog



has structure

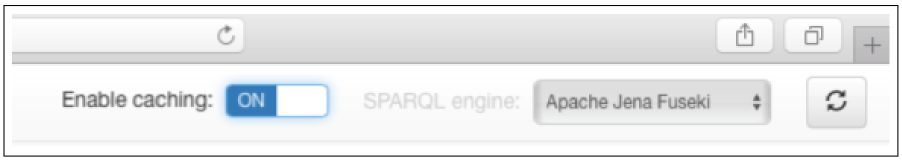
Broker

comprises tactic

Use Intermediary

Restrict Communication Paths

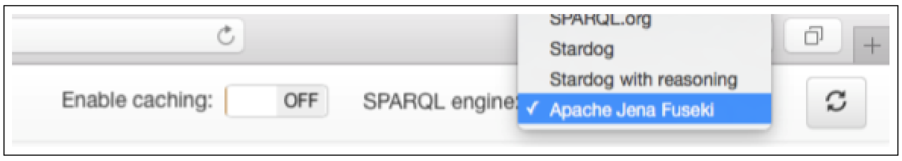
Figure A.6: Catalog item details



Enable caching: ON

SPARQL engine: Apache Jena Fuseki

(a) with caching enabled (engine switch disabled)



Enable caching: OFF

SPARQL engine:

SPARQL.org

Stardog

Stardog with reasoning

✓ Apache Jena Fuseki

(b) with caching disabled

Figure A.7: Screenshot of the user preferences options

Appendix B

Classification schema

The classification schema discussed in Chapter 2 is included in following pages. The scheme is divided into 10 pages, as depicted in Figure B.1.

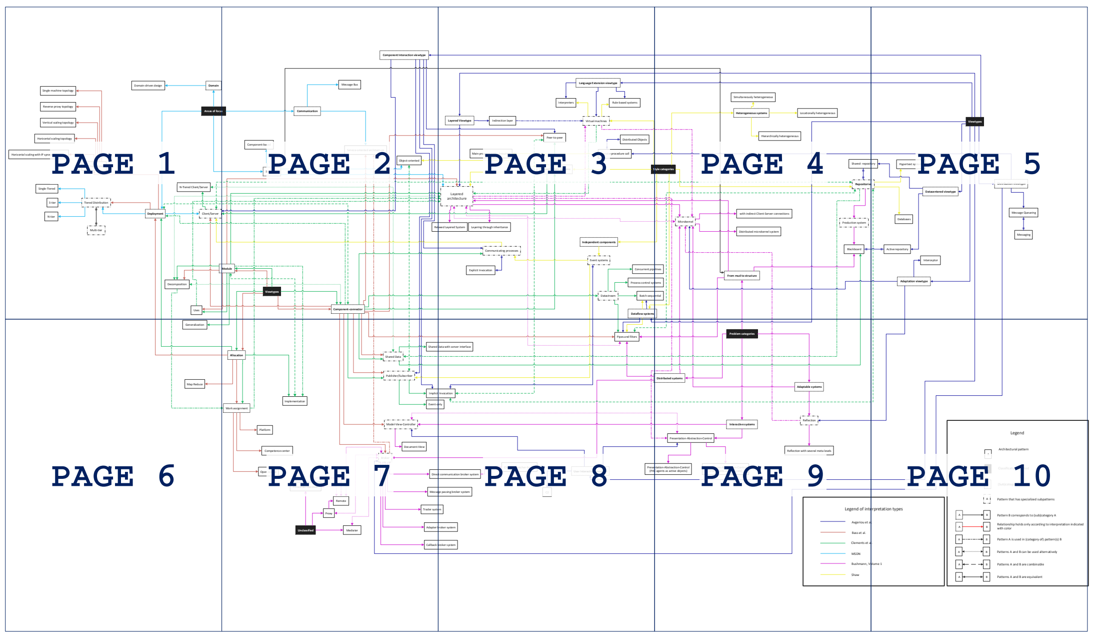


Figure B.1: Classification schema — overview

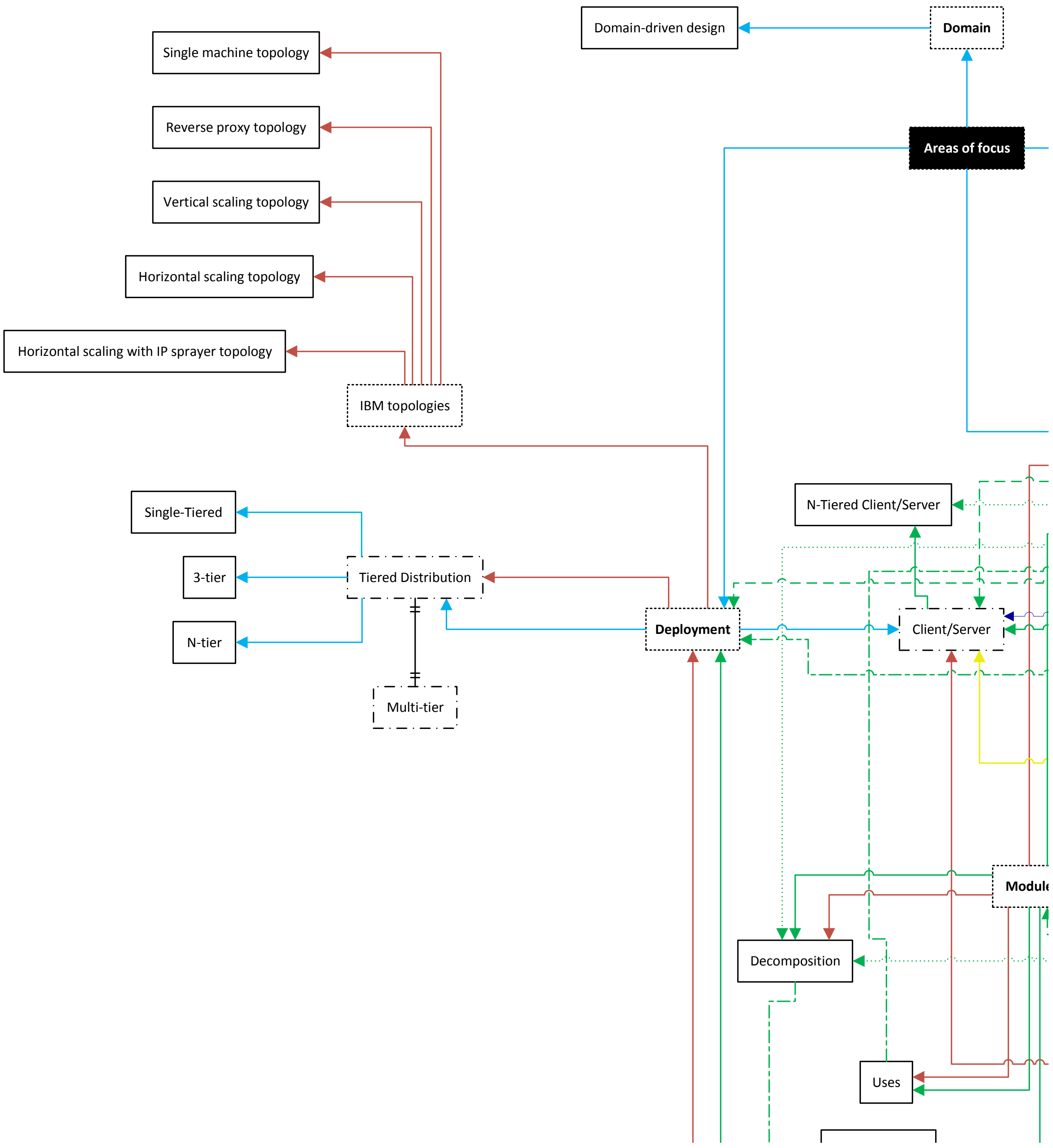


Figure B.2: Classification schema — page 1/10

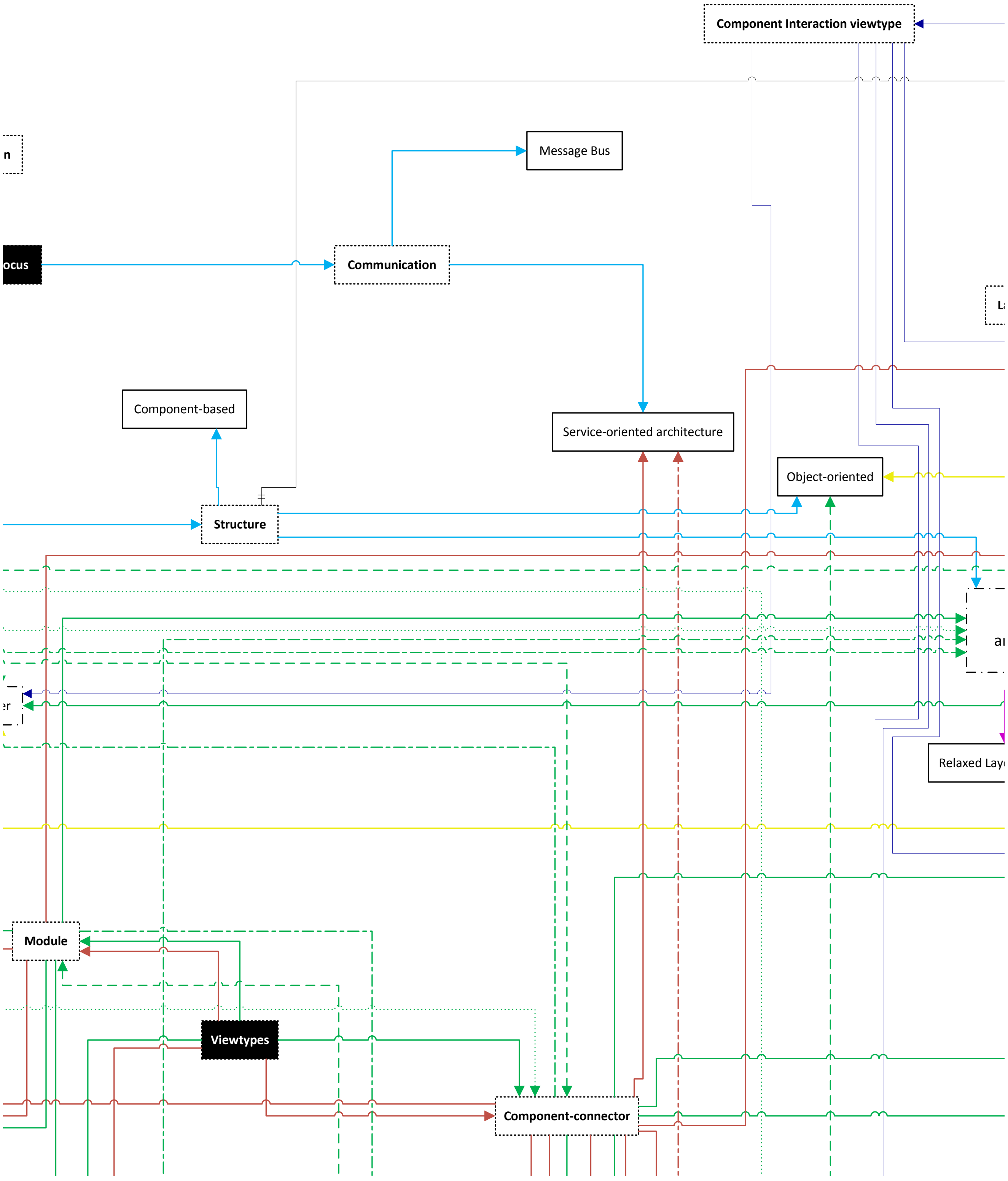


Figure B.3: Classification schema — page 2/10

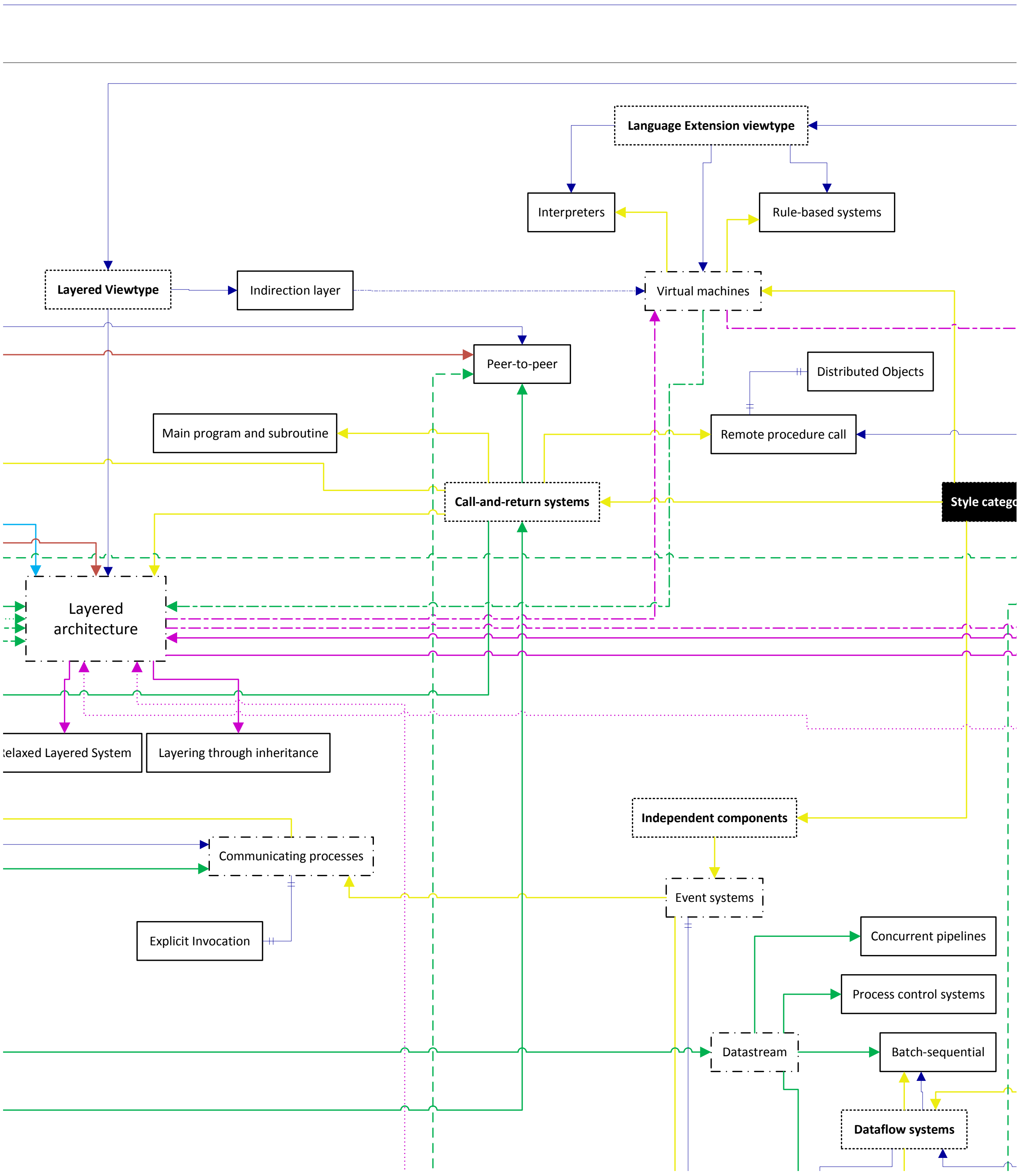


Figure B.4: Classification schema — page 3/10

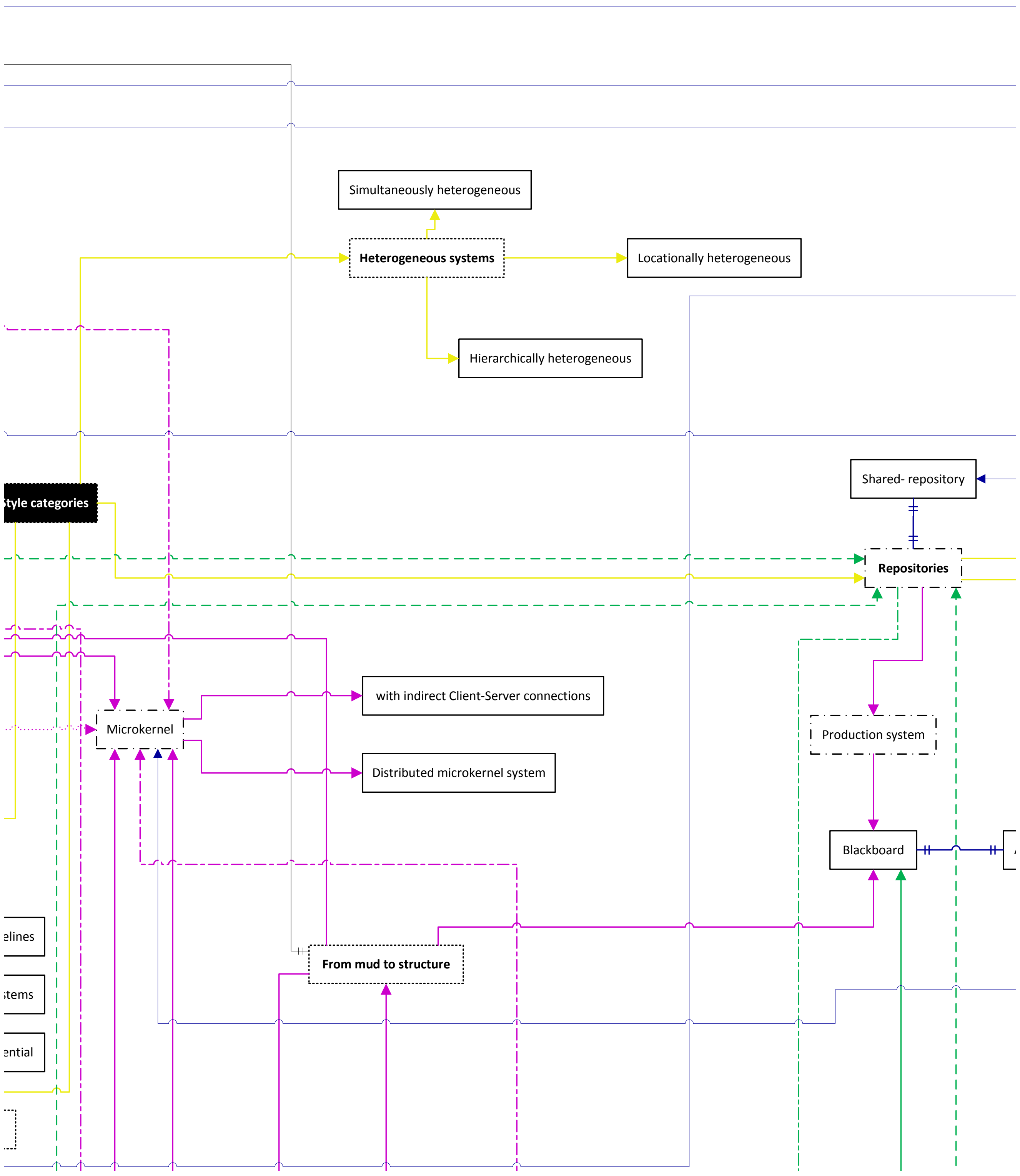

Simultaneously heterogeneous
Heterogeneous systems
Locationally heterogeneous
Hierarchically heterogeneous
Shared- repository
Style categories
Repositories
with indirect Client-Server connections
Microkernel
Production system
Distributed microkernel system
Blackboard
elines
stems
ential
From mud to structure

Figure B.5: Classification schema — page 4/10

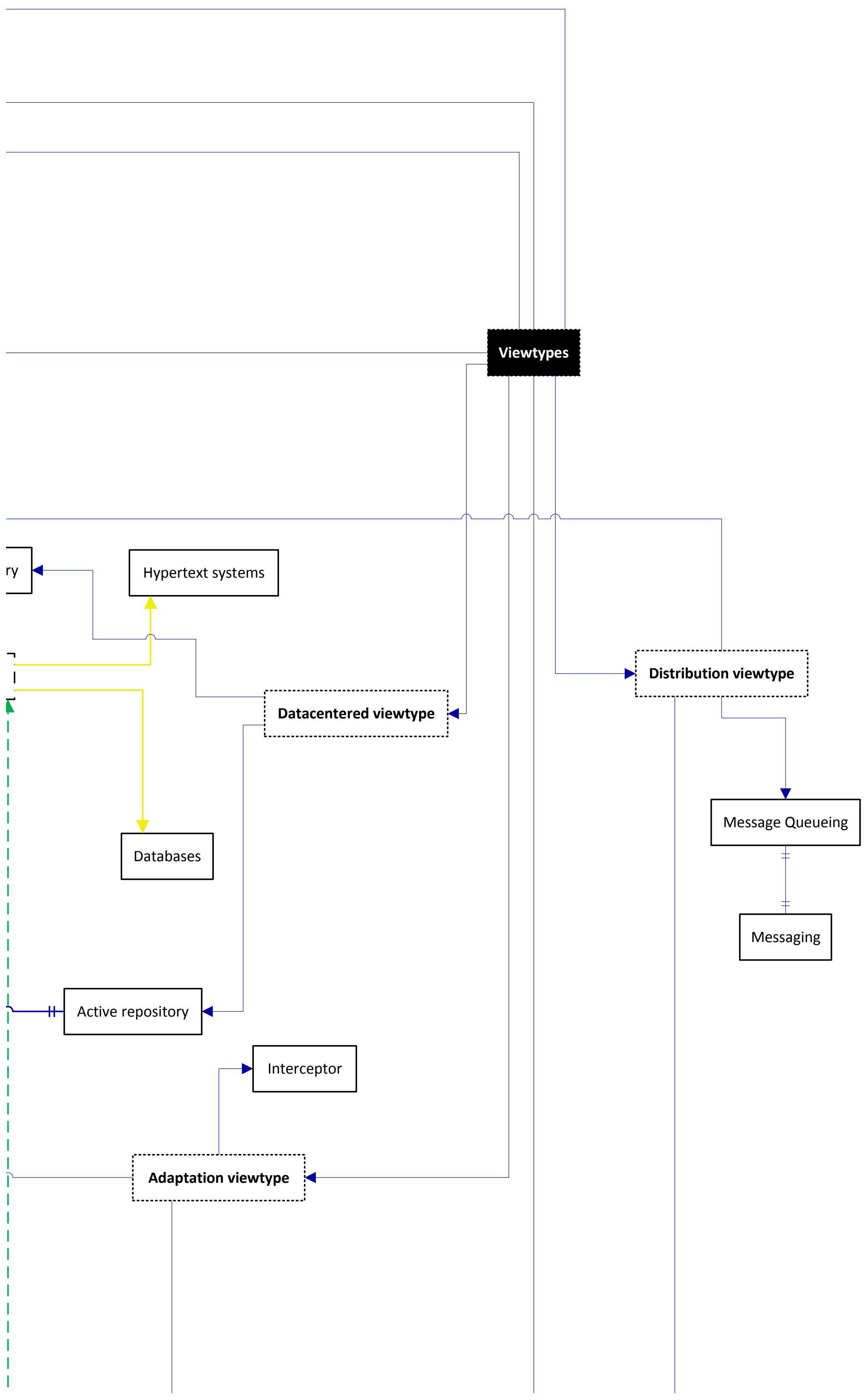


Figure B.6: Classification schema — page 5/10

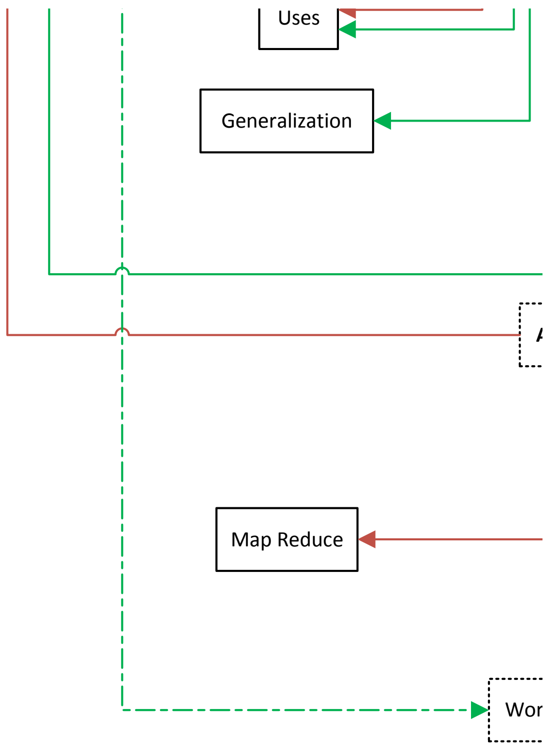


Figure B.7: Classification schema — page 6/10

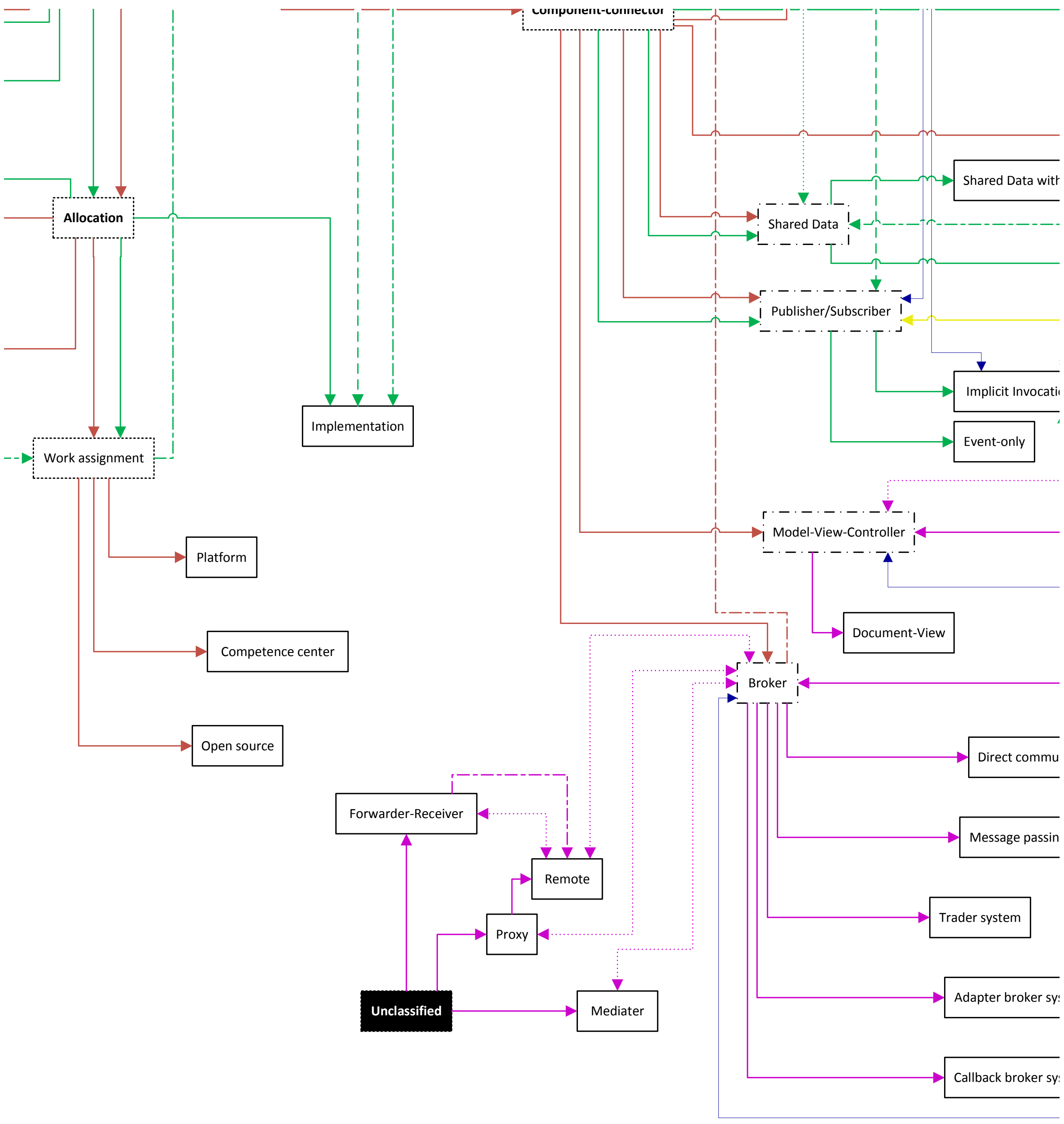


Figure B.8: Classification schema — page 7/10

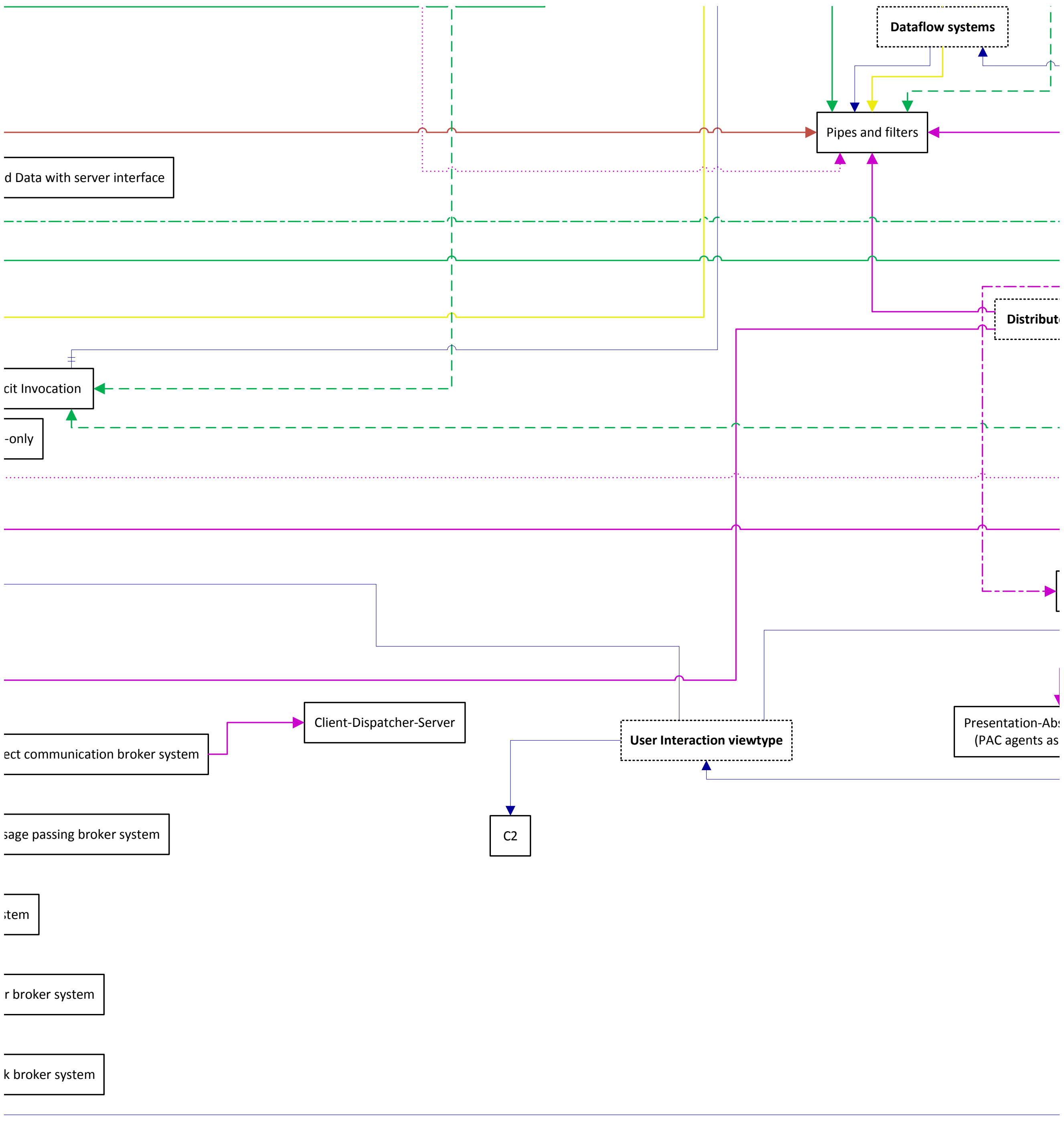
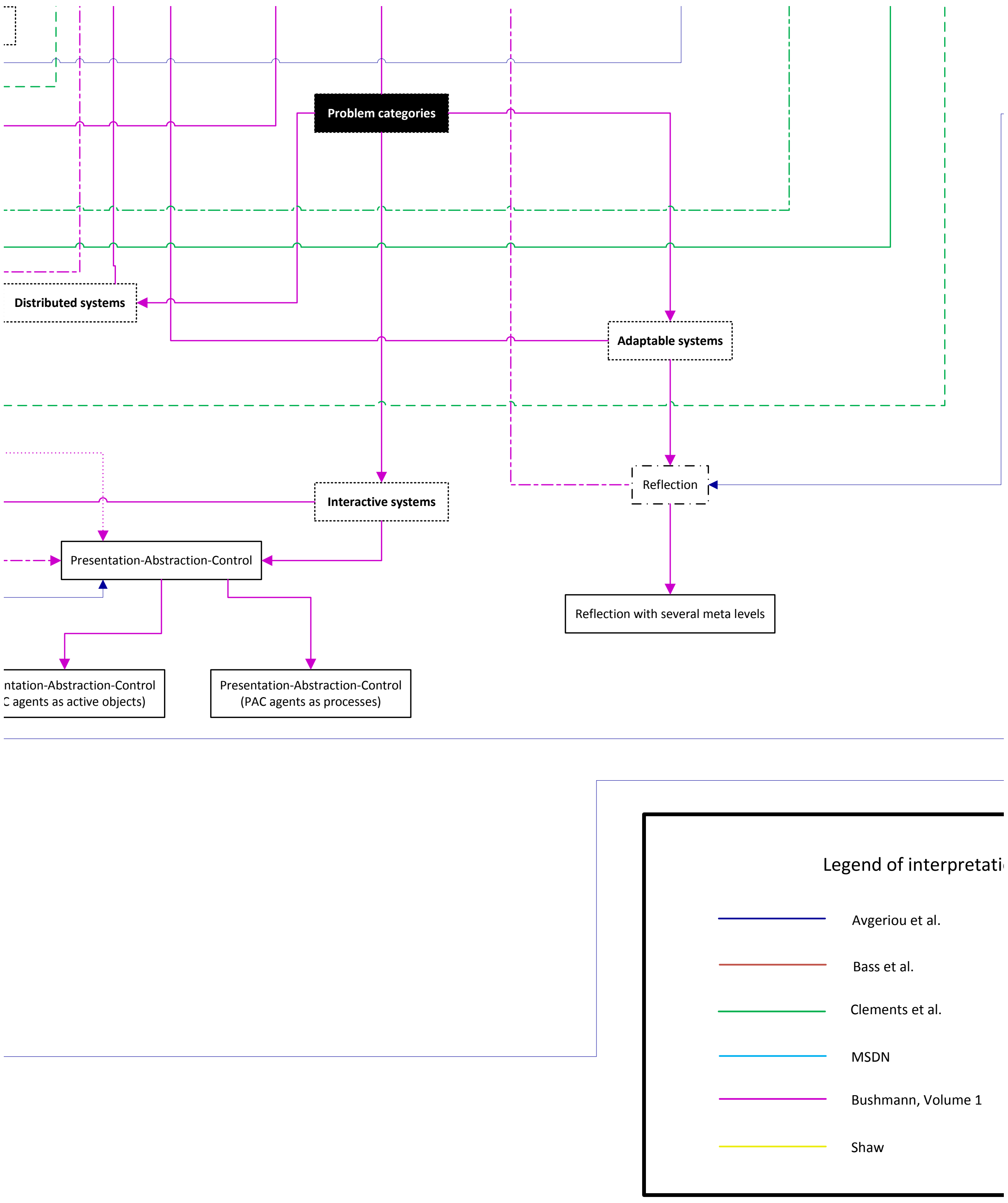


Figure B.9: Classification schema — page 8/10



Problem categories
Distributed systems
Adaptable systems
Reflection
Interactive systems
Presentation-Abstraction-Control
Reflection with several meta levels
ntation-Abstraction-Control
C agents as active objects)
Presentation-Abstraction-Control
(PAC agents as processes)
Legend of interpretati
Avgeriou et al.
Bass et al.
Clements et al.
MSDN
Bushmann, Volume 1
Shaw

Figure B.10: Classification schema — page 9/10

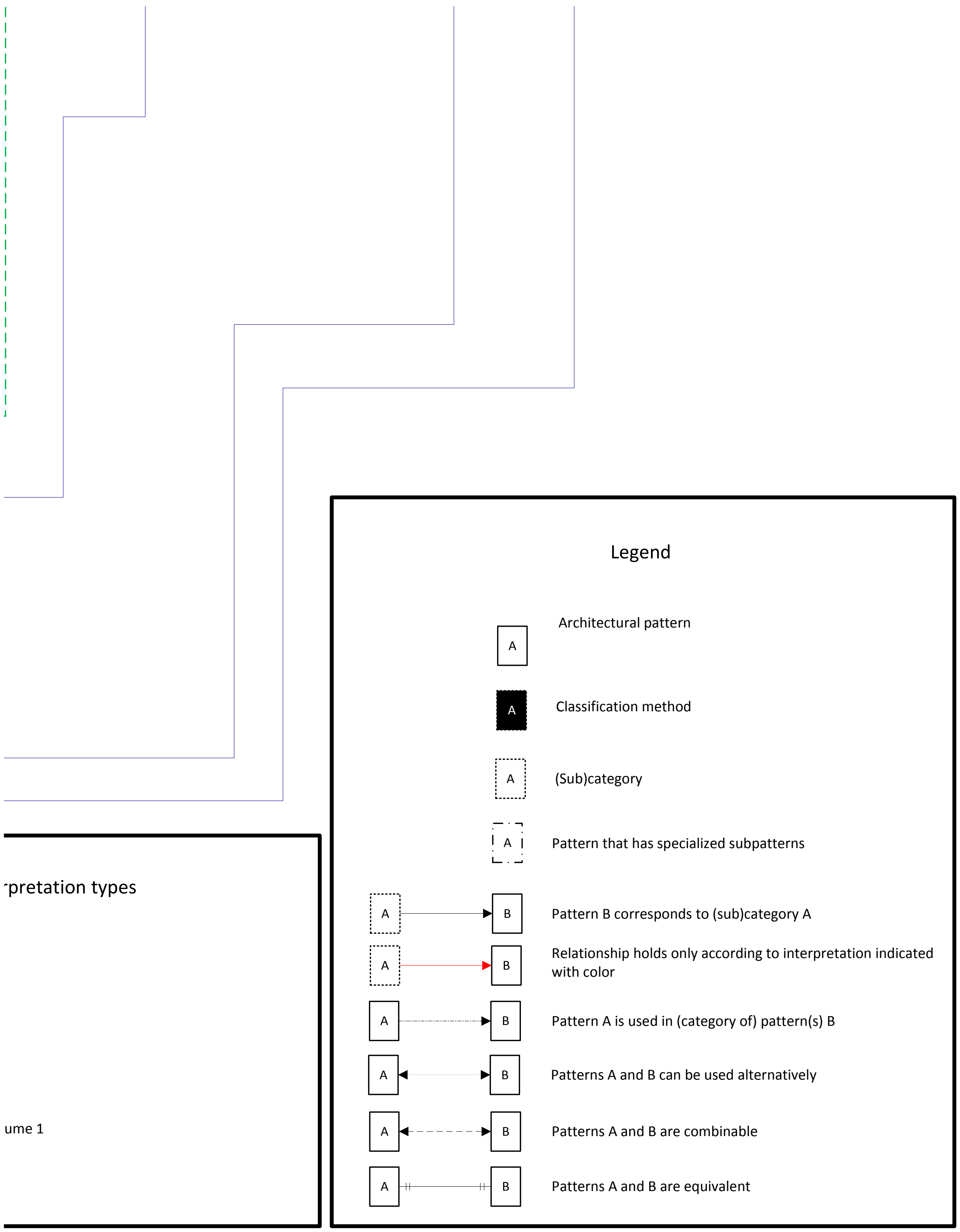

-pretation types
ume 1
Legend
A
Architectural pattern
A
Classification method
A
(Sub)category
A
Pattern that has specialized subpatterns
A
B
Pattern B corresponds to (sub)category A
A
B
Relationship holds only according to interpretation indicated with color
A
B
Pattern A is used in (category of) pattern(s) B
A
B
Patterns A and B can be used alternatively
A
B
Patterns A and B are combinable
A
B
Patterns A and B are equivalent

Figure B.11: Classification schema — page 10/10