Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

A Scalable Special-Purpose Metasearch Engine

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

unter der Leitung von

o. Prof. Dr. Georg Gottlob
und
Dr. rer. nat. Robert Baumgartner als mitwirkenden Assistenten

E184
Institut für Informationssysteme

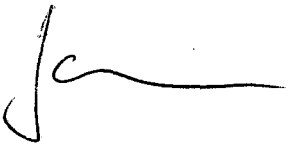
eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Mag. Ondrej Jaura

0327722

Wien, am 08.05.2006



Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Structure . . . 4

2 Internet Technologies **5**
- 2.1 XML Technologies . . . 5
  - 2.1.1 XML . . . 6
  - 2.1.2 XPath . . . 6
  - 2.1.3 XSLT . . . 7
  - 2.1.4 xQuery . . . 7
  - 2.1.5 Use in Metasearch Solution . . . 7
- 2.2 Hypertext Transfer Protocol . . . 8
- 2.3 HTML, XHTML . . . 8
  - 2.3.1 HTML . . . 8
  - 2.3.2 XHTML . . . 9
- 2.4 Forms . . . 9
  - 2.4.1 HTML Forms . . . 10
  - 2.4.2 HTML Form Submission . . . 11
  - 2.4.3 XForms . . . 12
- 2.5 Search Engine Forms . . . 14

3 Form Mapping **18**
- 3.1 Search Engines Interfaces Classification . . . 18
- 3.2 Interface Representation . . . 19
- 3.3 Matching . . . 20
  - 3.3.1 Semantic Matching . . . 21
  - 3.3.2 Weight-based Matching . . . 21
- 3.4 Global Interface . . . 23
  - 3.4.1 Merging Attribute Names . . . 24
  - 3.4.2 Merging Attribute Values . . . 24
  - 3.4.3 Creating Global Interface . . . 29
- 3.5 Global Result Presentation . . . 31

4 Existing Solutions and State of the Art **33**
- 4.1 Existing Meta Search Engines . . . 33
  - 4.1.1 Academic Projects . . . 33
  - 4.1.2 Commercial Projects . . . 36
  - 4.1.3 Lixto Project . . . 38

4.2 Lixto Suite . . . 38
4.3 Lixto Visual Wrapper . . . 39
4.3.1 Navigator . . . 39
4.3.2 Visual Wrapper . . . 40
4.3.3 Visual Developer . . . 42
4.3.4 Usage of Mozilla . . . 42
4.4 Transformation Server . . . 42
4.4.1 Components . . . 43
4.4.2 User Roles . . . 43
4.4.3 Hydra . . . 44
4.5 Lixto MetaSearch . . . 45
4.5.1 Requirements . . . 45
4.5.2 Current Status . . . 45
4.5.3 Mapping Framework . . . 50
4.5.4 Technology . . . 52
4.5.5 Intermediate Results . . . 53
4.5.6 Multiple Requests . . . 53
4.5.7 Session Identification Assignment . . . 54
4.5.8 Wizard Search Interfaces . . . 56
4.5.9 New Requirements . . . 56

5 Form Mapping Framework Design **58**
5.1 Form Descriptor . . . 58
5.2 Form Mapping . . . 59
5.3 Reverse Mapping . . . 60
5.4 Mapping Notation . . . 60
5.4.1 Complete Notation . . . 61
5.4.2 Semantic Notation . . . 62
5.4.3 Weight-based Rule Notation . . . 63
5.4.4 Combined Notation . . . 64
5.4.5 Value Mapping Notation . . . 64
5.5 Advanced Mapping . . . 64
5.5.1 List Mapping . . . 64
5.5.2 Date Mapping Notation . . . 66
5.5.3 Currency Mapping Notation . . . 66
5.5.4 Script Mapping Notation . . . 66
5.6 URL Mapping . . . 69
5.7 Navigation Mapping . . . 71
5.8 Comparison . . . 71
5.9 Mapping Strategies . . . 74
5.10 Execution Language . . . 74

6 Execution Framework Design **75**
6.1 Requirements . . . 75
6.1.1 Vocabulary . . . 75
6.1.2 Basic Requirements . . . 76
6.1.3 Advanced Requirements . . . 77
6.1.4 Optimisation Requirements . . . 77
6.2 Actors . . . 78
6.2.1 Request Producer . . . 78

6.2.2 Result Consumer . . . 78
6.2.3 System . . . 79
6.2.4 Supervisor . . . 79
6.2.5 Scenario Developer . . . 79
6.3 Use Cases . . . 79
6.3.1 Vocabulary . . . 79
6.3.2 Request Producer . . . 80
6.3.3 Result Consumer . . . 80
6.3.4 System . . . 82
6.3.5 Supervisor . . . 87
6.3.6 Scenario Developer . . . 87
6.4 Components . . . 90
6.4.1 Engine . . . 90
6.4.2 Manager . . . 91
6.4.3 User Interface . . . 91
6.4.4 User . . . 91
6.4.5 Administrator Interface . . . 93
6.4.6 Administrator . . . 93
6.4.7 Developer . . . 93
6.4.8 System Actor Components . . . 94
6.5 Components Design . . . 94
6.5.1 Components Relationship . . . 94
6.5.2 Components Intercommunication . . . 97
6.5.3 Engine Design . . . 97
6.6 Architecture . . . 102
6.7 Interconnection with Transformation Server Design . . . 103

7 Prototype Implementation 106
7.1 Technologies and Their Application . . . 106
7.1.1 Java . . . 106
7.1.2 Java Enterprise Edition . . . 107
7.1.3 Web Services . . . 107
7.1.4 Java Message Service . . . 110
7.1.5 Java Server Pages . . . 110
7.1.6 Java Server Faces . . . 111
7.1.7 Enterprise Java Beans . . . 111
7.1.8 Java Native Interface . . . 114
7.1.9 Databases . . . 114
7.1.10 Application Servers . . . 115
7.2 Overview . . . 115
7.3 Server . . . 116
7.3.1 Engine . . . 116
7.3.2 Database . . . 119
7.3.3 Octopus . . . 119
7.3.4 Navigator & Data Extractor . . . 120
7.3.5 Mapping Framework . . . 121
7.4 User . . . 122
7.5 Administrator . . . 122
7.6 Interaction . . . 124

8 Flight Metasearch Case Study **129**
8.1 Introduction 129
8.2 Requirements 129
8.3 Form Descriptors 130
8.3.1 SkyEurope 130
8.3.2 Ryanair 130
8.3.3 Global Search From 132
8.4 Mappings 133
8.5 Global Interface 133
8.6 Server Connection 134
8.7 Results 134
8.7.1 Extraction 134
8.7.2 Presentation 136
8.8 Conclusion 137

9 Conclusion **140**
9.1 Requirements Fulfilment 140
9.2 Further Development 141
9.3 Possible Application and Economic Impact 142

A Screenshots **144**

B Source Code **150**

List of Figures

2.1 XPath Example 7
2.2 HTML form tags example source 11
2.3 HTML form tags example 12
2.4 XForms presentation options 13
2.5 XForms main aspects 14
2.6 Google search form 14
2.7 Amazon search form 15
2.8 SkyEurope search form 16
2.9 Ryanair search form 17

3.1 Range numeric domain - one selection list. 26
3.2 Range numeric domain - one selection list and one text box. . . . 26
3.3 Range numeric domain - two text boxes. 27
3.4 Wizard type search form – first step 30
3.5 Wizard type search form – second step 30
3.6 Simple type search form 30
3.7 Global interface search form 31
3.8 Overview of scenario creating 32

4.1 Result pages navigator 39
4.2 Processing of Navigator and Visual Wrapper 41
4.3 Hydra as layer 44
4.4 Transformation Server Processing. 46
4.5 MetaSearch Server Processing. 46
4.6 MetaSearch work flow. 47
4.7 MetaSearch - login screen. 48
4.8 MetaSearch - administrator console screen. 49
4.9 Abstraction of local values 50
4.10 Global mapping 51
4.11 URI builder example 52
4.12 Success test results 55

5.1 Form descriptor 59
5.2 Form descriptor example 59
5.3 Mapping notation of common elements 61
5.4 Local elements mapping notation 61
5.5 Mmapping notation with normalisation 61
5.6 Example of complete mapping notation 62
5.7 Semantic list for mapping structure 62

5.8 Semantic mapping notation example 63
5.9 Weight-based rule notation structure 64
5.10 Combined notation structure 64
5.11 Full notation structure example 65
5.12 Date mapping notation structure 67
5.13 Currency mapping notation structure 67
5.14 Scripts list structure 68
5.15 Example of local script mapping notation 69
5.16 Example of local script mapping notation 70
5.17 Example of parameter script mapping notation 70
5.18 URL mapping 72
5.19 Navigation mapping 73

6.1 Use cases - request producer and result consumer. 81
6.2 Use cases - system. 82
6.3 Flowchart - accept request. 83
6.4 Flowchart - perform processing. 86
6.5 Use cases - supervisor and scenario developer. 88
6.6 Flowchart - develop scenario. 89
6.7 Components - system actor components. 95
6.8 Components - separation and relationships. 96
6.9 Components intercommunication - no cached. 98
6.10 Components intercommunication - cached. 99
6.11 Components intercommunication - stopped. 100
6.12 Engine schema 102
6.13 Simplified architecture design. 103
6.14 Complex architecture design. 104

7.1 J2EE Architecture Diagram 108
7.2 Stateless session bean life cycle 112
7.3 Stateful session bean life cycle 112
7.4 Entity bean life cycle 113
7.5 Message-driven bean life cycle 114
7.6 Architecture implementation. 117
7.7 Engine class diagram 118
7.8 Database Entity Relationships 119
7.9 Octopus package class diagram 120
7.10 Mapping package class diagram 121
7.11 User and admin package class diagram 123
7.12 Administrator management console page 124
7.13 Sequence diagram for Flight Search 125
7.14 Data flow diagram for Flight Search 128

8.1 Form descriptor for SkyEurope search form 131
8.2 Form descriptor for Ryanair search form 132
8.3 Form descriptor for global search form 133
8.4 Mapping for flight metasearch 134
8.5 Flight search forms 135
8.6 Flight output example skeleton 136
8.7 Flight Search result page with started search process 137

8.8 Flight Search result page with one result 137
8.9 Flight Search result page with two results 138
8.10 Flight Search result page with finished search process 138

A.1 Intermediate result example - start 144
A.2 Intermediate result example - start 145
A.3 Visual Designer - wrapping screen. 146
A.4 Transformation Server - developer interface screen. 147
A.5 MetaSEEk search form 148
A.6 KartOO graphical results presentation 149
A.7 Vivísimo clustered results 149

B.1 Input entity bean source code 150
B.2 ParameterValue entity bean source code 150
B.3 SearchResult entity bean source code 151
B.4 Request session bean source code 151
B.5 Heartbeat session bean source code 152
B.6 Response session bean source code 153
B.7 Result JSP page source code 153
B.8 Result web service interface source code 154
B.9 Result web service interface source code 154

List of Tables

4.1 Multiple requests test results 54
4.2 Success test results 55

Kurzfassung

Gleich nach dem Mailen, ist das Suchen die beliebteste Tätigkeit im Internet und Suchmaschinen sind das Werkzeug, mit dem man an die unermesslichen Informationsmengen im World Wide Web herankommt. Eine Lösung, die auf der Zusammenführung von Suchergebnissen mehrerer Suchmaschinen basiert – wird als Metasuche bezeichnet. Eine Lösung, die spezialisierte Suchmaschinen und Metasuchmaschinen kombiniert, bezeichnet man als spezialisierte Metasuche oder Spezial-Metasuche. Eine Spezial-Metasuche ist in der Lage Suchergebnisse von nachgestellten Datenbanken zu integrieren, die über konkrete spezialisierte Suchmaschinen verfügbar sind.

Ein gutes Beispiel dafür, wie Technologie einem Endkunden dabei hilft die zweckmäßigsten Angaben zu finden – sind Flugbuchungen. Durch die Integration von Suchergebnissen von mehreren Buchungssystemen der Fluggesellschaften können die Kunden vergleichen um so den am besten geeigneten und günstigsten Flug unter mehreren Fluggesellschaften zu finden. Ohne eine spezialisierte Metasuchlösung, müsste der Endkunde die Buchungssysteme jeder einzelnen Fluggesellschaft prüfen, d.h. jede spezialisierte Suchmaschine. Eine durchdachte Metasuchlösung ermöglicht auch Währungskonversionen, die Sortierung der Suchergebnisse und das Kombinieren von mehreren Flugrouten verschiedener Gesellschaften.

Diese Arbeit entwickelt eine neuartige spezialisierte Metasuchmaschine, die für jeden Geschäftszweck anwendbar ist; sie präsentiert den theoretischen und technischen Hintergrund, den Entwurf wie auch die für einen kommerziellen Einsatz geeignete Umsetzung. Der technische Hintergrund befasst sich mit solchen Gebieten, wie der Mapping-Theorie; der Entwurf zeigt Einzelheiten des Mapping- und Ausführungsrahmens und bei der Umsetzung werden zur Erreichung einer robusten und skalierbaren Lösung unternehmenseigene Technologien eingesetzt.

Die beigefügte Fallstudie ist eine Flug-Metasuche, eine Spezialmetasuche, die die Suchergebnisse der Buchungssysteme von Fluggesellschaften integriert.

Abstract

Search is after e-mail is the second most popular activity on the Internet and search engines are tools to reach the extensive amount of information on the World Wide Web. A solution based on the aggregation of results from more search engines, that can lead to better quality and higher number of results, is called *metasearch*. A solution that combines specialised search engines and metasearch engines is called *specialised metasearch* or *special-purpose metasearch*. A specialised metasearch solution is able to integrate results from backend databases that are available through particular specialised search engines.

A good example of how this technology helps an end customer to find the most appropriate data is the flight booking area. The integration of results from more airline flight booking systems allows users to compare and find the most suitable and cheapest flight from a number of airlines. Without a specialised metasearch solution an end customer would have to check every single airline flight booking system, i.e. each specialised search engine. A sophisticated metasearch solution can also provide features such as currency conversion, sorting of results and combining more routes from different airlines.

This thesis develops a novel special-purpose metasearch engine that can be used for any business purpose, presenting its theoretical and technical background, design and implementation suitable for commercial use. The technical background embraces areas such as mapping theory; the design shows details of the mapping and execution frameworks, and the implementation uses enterprise technologies to achieve a robust and scalable solution.

The case study included is – a flight metasearch, a special-purpose metasearch that integrates results from airline booking systems.

Chapter 1

Introduction

1.1 Motivation

Search engines are tools to reach the extensive amount of information on the World Wide Web. Recent studies mentioned in the [ZMW+05] paper show that search after e-mail is the second most popular activity.

Common search engines search static web pages with a simple and generic search interface. They do not search specific data stored in backend databases, also known as the deep web, which are estimated to be 500 times larger than the static web, as mentioned in the [Ste01] paper. The scale of searched backend databases raises various technical problems.

Moreover, users are not able to retrieve the required information because of the too generic search interface provided by common search engines.

A solution based on the aggregation of results from more search engines is called *metasearch*. Each search engine has its strengths and weaknesses and the integration of results from more search engines can lead to the better quality and higher number of results. For example there are metasearch engines that combine results from the most popular search engines such as Google, Yahoo and MSN.

The search interface of a metasearch engine is similar to its search engines.

A solution that combines both worlds of specialised search engines and metasearch engines is called *specialised metasearch* or *special-purpose metasearch*. It is a solution that applies the metasearch approach to an area of specialised search engines such as flight booking systems or car rental systems. A specialised metasearch solution is able to integrate results from backend databases that are available through particular specialised search engines. A search interface is similar to the specialised search interfaces of the search engines used.

A good example of how this technology helps an end customer to find the most appropriate data is the flight booking area. The integration of results from more airline flight booking systems allows to compare and find the most suitable and cheapest flight from a number of airlines. Without a specialised metasearch solution an end customer would have to check every single airline flight booking system, i.e. a specialised search engine. A metasearch solution

can also provide features such as currency conversion, the sorting of results and combining more routes from different airlines.

This thesis describes the special-purpose metasearch engine, *Snorri*[1], which can be used for any business purpose, its technical background, design and implementation suitable for commercial use. The technical background embraces areas such as mapping theory; the design shows details of the mapping and execution frameworks.

It vastly improves and extends the existing Lixto MetaSearch version by eliminating limitations such as synchronous provision of results, complicated creation and deployment of scenarios, and introduces caching, scalability and load balancing.

The proposed solution allows a user to be up-to-date and to choose the most profitable option at any time – it offers an instant economic advantage on a real-time enterprise level.

1.2 Structure

This thesis consists of seven chapters:

The *Internet Technologies* chapter describes technical areas necessary for a metasearch solution. From the low-level HTTP requests to a description of an HTML form.

The *Forms Mapping* chapter explains a theoretical background of an item mapping from local interfaces to the global interface and vice versa.

The *Existing Solution and State of the Art* chapter summarizes possibilities, advantages and disadvantages of the existing solutions.

The *Form Mapping Framework Design* chapter shows how to use the above mentioned theory and techniques to create and design a form mapping framework.

The *Execution Framework Design* chapter describes requirements, use cases, flowcharts and class diagrams of the execution framework design.

The *Prototype Implementation* chapter shows technologies used for the a prototype implementation. Interesting parts of the code are also shown.

The *Flight Metasearch Case Study* chapter contains a case study where the proposed metasearch solution is used for a flight search service.

The *Conclusion* chapter summarizes achieved goals, next steps and possible improvements.

[1] Snorri Thorfinnsson, born sometime between 1005 and 1013. He was purported to have been born in Vinland, a Viking settlement in North America, possibly making him the first European to born in North America. For more details see http://en.wikipedia.org/wiki/Snorri_%C3%9Eorfinnsson Wikipedia page.

Chapter 2

Internet Technologies

The metasearch world is a part of the Internet universe. It consists of Internet sources such as HTML pages that are produced by a server that receives requests. A request or a page is transferred via the HTTP protocol. Necessary data is extracted to XML documents, that are changed and concatenated by XSL transformations in to the final form as presented to a user.

All technologies that are important and necessary for a metasearch solution are mentioned in the following sections.

The first section explains XML technologies as a malleable exchange medium between processes. The second section shows the main protocol between a server and a client in the WWW world, HTTP. The third part describes the main element of the WWW world, an HTML page. The next section focuses on an HTML form as the entry point for user requests that are substituted with a metasearch solution. The last section shows real examples of search forms and points out typical and problematic parts.

2.1 XML Technologies

XML technologies represent a set of standards such as data format, transformation and query specified by the World Wide Web Consortium (W3C).

The data format is represented by XML. Nowadays it is an industrial standard for exchanging information between processes. It is described in Section 2.1.1.

An increasing amount of data represented by the XML data format has caused the introduction of query languages for the XML world – XPath and XQuery, that are described in Section 2.1.2 and Section 2.1.4.

A transformation functionality is represented by XSLT described in Section 2.1.3.

The usage of these technologies in the Metasearch world is explained in Section 2.1.5.

2.1.1 XML

Extensible Markup Language, XML, is a language for pure structural description. It is organised as a tree; one tree is called an XML document.

XML represents a possibility of data organisation in a tree structure. There are many languages in XML that describe different areas: from MathML, a mathematical notation suitable for web pages and WSDL, a description of web services to languages for the description of relationships of elements, such as FOAF.

XML is a restricted form of SGML, the Standard Generalized Markup Language. The construction of XML documents conforms to SGML documents. It was developed in 1996.

The specification [Con04a] says: *"XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure."*

Origin and Goals

The design goals for XML, described in the [Con04a] specification, are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.

2.1.2 XPath

XPath, described in the [Con05b] specification, addresses a node or a set of nodes in an XML tree. XPath uses a compact syntax for its values and has many possibilities to express an address.

For example, see Figure 2.1, where the node on the second line can be addressed with the `/library/book[1]` XPath expression, all books can be addressed with the `//book` XPath expression, where it does not matter how many parents have a book node.

```
1 <library>
2         <book>A</book>
3         <book>B</book>
4 </library>
```

Figure 2.1: An XPath example.

It represents the core of the XSLT technology, which makes it possible to specify a node or a set of nodes.

With the new XSTL2 a new version of XPath, named XPath2, was published. New features are described in the [Con05d] specification.

2.1.3 XSLT

XSLT is a declarative rule language for the transformation of a source XML tree into a result tree. The result tree can be completely different from the structure of the source tree. In the process of constructing the result tree, elements from the source tree can be filtered, reordered and modified and added to the result tree.

XSLT 1.0, described in the [Con99] specification, was published in November 1999. The new version 2.0 was published in 2005 and represents a significant increase in the capability of the language. XSLT2 is described in the [Con05d] specification.

2.1.4 xQuery

The ability to query XML data becomes more important with increasing amounts of stored and exchanged information using XML. A flexibility in representing different kinds of data from various sources is one of the main strengths of XML. To exploit this flexibility, an XML query language must provide features for retrieving and interpreting information from these diverse sources.

There are the following differences between xQuery and XSTL:

- xQuery is more human-legible. It is mainly based on the standard query language for databases, SQL.
- it is a functional language.
- it is strongly typed.

xQuery operates on the abstract, logical structure, known as the data model, of an XML document, rather than its surface syntax.

xQuery Version 1.0, described in the [Con05c] specification, is an extension of XPath Version 2.0. Any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages.

2.1.5 Use in Metasearch Solution

For a metasearch solution the XML data format is a suitable exchange and storage medium for all parts, from internal structures to user data.

The natural variability of the XML data format with rich support for transformations and querying is a better data format in contrast to a rich querying but fixed structure of relational databases or with a high variability of own binary or text format but a lack of transformation and query standards and tools.

Various user data can be generally transformed by XSLT and their processing of them can use the XPath and XQuery query languages.

High variability of XML standards on the other hand requires more processor and memory resources. The processing of large XML documents may require special handling: optimised algorithm for detecting duplicates is described in the [Nau04] paper, the [KBNK02] paper shows improved efficiency of the XPath query language using indices, and the [GKP02], [GKP03b] and [GKP03a] papers describe and improve an XPath query evaluation.

The proposed metasearch solution in further chapters resolves this problem by processing results for each Internet source separately.

2.2 Hypertext Transfer Protocol

The Hypertext Transfer Protocol, *HTTP* is the basic communication protocol in the web world. It has been in use since 1990.

The [W3C99b] specification describes HTTP as: *"It is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred."*

It is basically a request–response protocol that is sent via TCP/IP connection. A client sends a request to a server in the form of a request method, URI, and a message. The server responds with a status line, a success or error code, and a message. Request methods are described in Section 2.4.2.

2.3 HTML, XHTML

In the following sections markup languages designed for the creation of web pages are presented.

2.3.1 HTML

The HyperText Markup Language, or *HTML*, is a form of a markup language that is used to create and publish hypertext documents. Viewing of documents is provided by specialized rendering applications called HTML user agents mainly known as web browsers. It is a non-proprietary format built on an SGML grammar.

HTML provides various kinds of meta-data and rendering hints that are included in a document. The meta-data include information about the document title, author and keywords, structure elements like headings, paragraphs, lists

and information that allows the document to be linked to other documents. The last feature allows to form a *hypertext web.*

The rendering hints provide features such as text decoration, insertion of images and form definition, which is the main area of interest and will be explained in detail in the following sections.

HTML is standardised by the World Wide Web Consortium, the latest is the HTML 4.01 [W3C99a] specification that includes minor revision of the HTML 4.0 specification. It was published in December 1999.

The HTML 4.0 *cleaned up* the standard by marking some elements as deprecated. It also adopted browser-specific element types and attributes. It was published in December 1997.

The HTML 3.0 specification was proposed by W3C in 1995 and provided many new capabilities. Unfortunately it was too complex and due to lack of browser support it was discontinued. The next HTML 3.2 specification, that was accepted, adopted element types and attributes of Netscape and Mosaic web browsers. It was published in January 1997.

The HTML 2.0 was created as the first formal specification. The version number 2.0 helped to distinguish it from many unofficial 1.0 *quasi-standards.*

HTML can be generated on demand by a server-side scripting system such as PHP, JSP or ASP.

2.3.2 XHTML

Extensible HyperText Markup Language, XHTML, is a family of document types and modules that reproduce, subset, and extend HTML 4.0. The [Con00] specification says: *"XHTML family document types are XML based, and ultimately are designed to work in conjunction with XML-based user agents."*

The main differences between HTML and XHTML are:

- XHTML is an XML based language. An application that processes an XHTML page can be simpler. An application that processes an HTML page has to be able for example, to complement missing closing tags on the right position, which is a non-trivial process.
- XHTML defines several levels, where an application or a designer can decide how complicated and complex pages will be accepted or created, e.g. with or without frames.
- XHTML elements must be properly nested, documents must be well formed, tag and attributes names must be written lower-case, empty elements must be terminated and attribute values must be quoted.

2.4 Forms

An area on a page designed for exchanging data between a server application and a user is called a *form.* It is usually a rectangle area visibly separated from other page elements and it contains at the bottom or on the right side a set of

buttons. One of the buttons is a button that submits entered data.

A form is the most important part of a page for a metasearch solution. As a user can enter data, also a metasearch system is able to use a form to enter its data to retrieve information from a server.

2.4.1 HTML Forms

HTML and XHTML have specific tags for defining a form and its elements. See Figure 2.2 for a form example source code and see Figure 2.3 for a form example visual representation.

The form tags are described in the following list:

- Form. The root tag of a form is the *form* tag. It is shown in Figure 2.2 in line 1. It specifies the request method in the *method* attribute and a URI where entered data should be submitted in the *action* attribute. It does not affect the graphical layout, that is, actually, a form on an HTML page can be spread over more visually separated areas.

- Input. The tag that represents an input element is the *input* tag. The *type* attribute specifies the type, the *name* attribute specifies the name, and the *value* attribute the value of an input element.

  There are the following input element types:

  - Text. The *text* input type represents a textual value that is displayed as an edit box where any text can be typed. For example, see Figure 2.2, line 2.
  - Radio. The *radio* type can be used only for a set of input elements with the same name, because it represents a choice between values of the elements with the same name where only one can be selected. For example, see Figure 2.2, lines 3 and 4.
  - Checkbox. The *checkbox* type can be used alone or as a set of input elements with the same name, because it represents a choice between values of the elements with the same name where none, one or more can be selected. For example, see Figure 2.2, line 5.
  - Submit and Reset. The *submit* and *reset* types are visually represented as a button. The submit type emits the submission process, for more details see Section 2.4.2. The reset type resets the form, i.e. it resets a value of all input elements. For example, see Figure 2.2, lines 12 and 13.
  - Hidden. The *hidden* type is a special type that allows the adding to a form technical data used by an application that produced the page and will receive the submitted data. An input element with the hidden type is not visible. For example, see Figure 2.2, line 6.

  The other types such as *password* or *file* are described in the [W3C99a] specification.

- Select and Option. The tag *select* is a special type of *input* tag. It represents a drop down box, where its values are defined with the *option* tag.

By default only one value can be selected, when the *multiple* attribute is added to the *select* tag, none or more values can be selected. For example, see Figure 2.2, from line 7 to 10.

- Textarea. The *textarea* tag is a special type of *input* tag. It represents an area where a text on more lines can be written, such as a remark. For example, see Figure 2.2, line 11.

```
1  <form action="process.jsp" method="get">
2  Surname: <input type="text" name="surname"/>
3  <input type="radio" name="sex" value="male"/>Male
4  <input type="radio" name="sex" value="female"/>Female
5  <input type="checkbox" name="license"/>I have a driving license
6  <input type="hidden" name="authorised" value="1"/>
7  Select a car type: <select name="car">
8          <option value="normal">Normal
9          <option value="luxurious">Luxorious
10 </select>
11 A remark: <textarea rows="2" cols="20"></textarea>
12 <input type="submit" value="Submit"/>
13 <input type="reset" value="Reset"/>
14 </form>
```

Figure 2.2: HTML form tags example source code. Formatting tags are excluded for clarity.

2.4.2 HTML Form Submission

Values from the completed form are transferred to a server in a process called *submission.*

The form data can be transferred in two ways specified by the requested method that is set in the *method* attribute of the *form* tag:

- Get Method. Form data are appended to the URI specified in the *action* attribute. Each parameter appears in a request as a name–value pair, where the equals sign '=' is the delimiter between the name and value parts, e.g. *name=John.* Pairs are delimited by the and sign '*&*' and the first pair is delimited by the question mark '*?*' from the previous part of URI, e.g. *http://server/process.jsp?name=john&surname=smith.* The created URI is sent to the processing agent.

  The [W3C99a] specification says: *"This method should be used when the form is idempotent (i.e., causes no side-effects). Many database searches have no visible side-effects and make ideal applications for the get method."*

  The length of URI is limited in browsers; large data should be sent with the *post* method.

  The set of allowed characters in a URL is limited. Other characters have to be *URL encoded,* i.e. a forbidden character is replaced with a percent

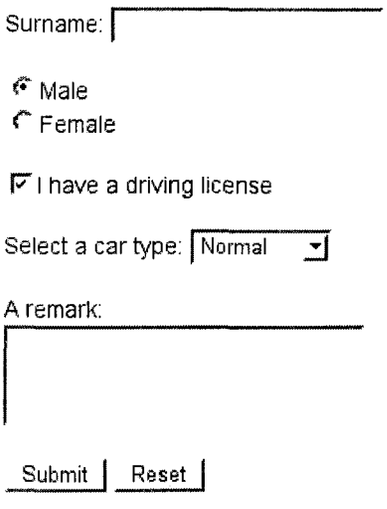


Figure 2.3: HTML form tags example.

'*%*' sign, as a prefix, and its hexadecimal numeric value defined in the ASCII table, e.g. a space is replaced with '*%20*'.

- Post Method. Form data are sent to the processing agent included in the body of the sent message.

  The [W3C99a] specification says: *"If the service associated with the processing of a form causes side effects (for example, if the form modifies a database or subscription to a service), the post method should be used."*

  This method is also used when a form contains large data such as files, i.e. a form contains an input element with the *file* type.

2.4.3 XForms

XForms is a new form standard proposed by W3C organisation that improves the current minimal support for forms.

HTML Web forms do not separate the purpose from the presentation of a form. The [Con05a] specification says: *"XForms, in contrast, are comprised of separate sections that describe what the form does, and how the form looks. This allows for flexible presentation options, including classic XHTML forms, to be attached to an XML form definition."*

The schema in Figure 2.4 illustrates how a single XML form definition, called the XForms Model, has the capability to work with a variety of standard or proprietary user interfaces.

The XForms User Interface provides a standard set of visual controls that replace common XHTML form controls. These form controls are directly usable inside XHTML and other XML documents such as SVG.

The XForms [Con06c] page says: *"An important concept in XForms is that forms collect data, which is expressed as XML instance data. Among other duties, the XForms Model describes the structure of the instance data. This is*

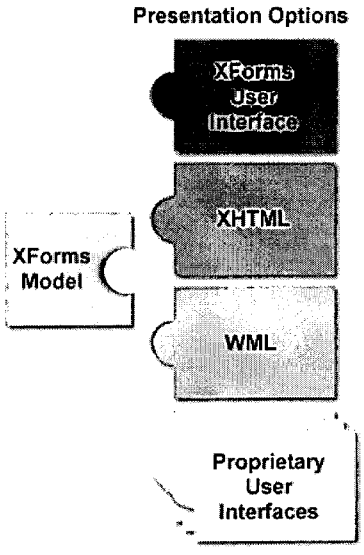


Figure 2.4: The presentation options of XForms, as described on the [Con06c]page.

important, since like XML, forms represent a structured interchange of data. Workflow, auto-fill, and pre-fill form applications are supported through the use of instance data."

The XForms Submit Protocol defines how XForms send and receive data, including the ability to suspend and resume the completion of a form.

The illustration in Figure 2.5 summarizes the main aspects of XForms, as described on [Con06c] page.

Key Goals of XForms

XForms has the following key goals that are mentioned on [Con06c] page:

- Support for handheld, television, and desktop browsers, plus printers and scanners
- Richer user interface to meet the needs of business, consumer and device control applications
- Decoupled data, logic and presentation
- Type checking
- Improved internationalization
- Support for structured form data
- Advanced form logic
- Multiple forms per page, and pages per form

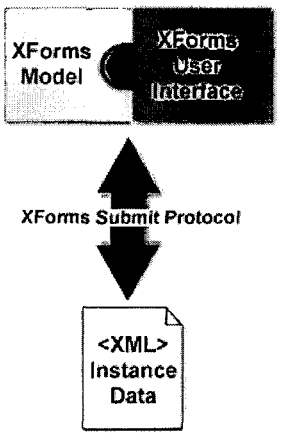


Figure 2.5: The main aspects of XForms, as described on [Con06c] page.

- Suspend and Resume support
- Seamless integration with other XML tag sets

2.5 Search Engine Forms

For a metasearch solution search forms are the main way to get data. This section shows technical details and interesting parts of important search engines in various areas such as book shopping or flight ticket ordering.

Google

The best known search engine for the search of a text phrase is *Google*.

Its search [Goo06c] form, shown in Figure 2.6, is simple, it contains only a plain HTML tag without any JavaScript code calls necessary to submit form data.

A metasearch solution can produce a URL directly or emulate user actions to enter form data. Basically there are no pitfalls.

```
1 <form action=/search name=f>
2 <input maxlength=2048 size=55 name=q value=""
3         title="Google Search">
4 <input type=submit value="Google Search" name=btnG>
```

Figure 2.6: Interesting parts of the Google [Goo06c] HTML search form.

Amazon

Amazon is one of the biggest Internet shops. It contains several search forms; a universal search is on the top of all pages, some specialised pages contain a

special search form.

Its universal search [Ama06] form, shown in Figure 2.7, is a simple HTML form without any JavaScript method calls to submit form data. One speciality of the form is that it submits data with the *image* input type, see line 6.

A metasearch solution can produce a URL directly or emulate user actions to enter form data. Basically there are no pitfalls.

```
1 <form method="get"
2         action="search-handle-url/ref=br_ss_hs/102">
3 <input type="hidden" name="platform" value="gurupa" />
4 <input type="text" id="twotabsearchtextbox"
5         name="field-keywords" value="" size="25"/>
6 <input type="image" src="images/go.gif"
7         value="Go" name="Go"/>
```

Figure 2.7: Interesting parts of the Amazon [Ama06] HTML search form.

SkyEurope

The *SkyEurope* airline offers cheap flight tickets in Europe. Its booking search form is a standard flight booking form, there is just one difference: it does not distinguish between adults and children. It contains only two categories: adults and infants, i.e. the age is from 0 to 24 months. A common form contain three categories: adults, children, i.e. under 16 years, and infants.

This search [Sky06a] form, shown in Figure 2.8 contains one problematic point – the *__VIEWSTATE* parameter value, see line 3. It is generated per request, i.e. every generated booking search page has its own specific value in this parameter.

A metasearch solution that sends a URL directly has to download the booking page, extract a value of the parameter and use it. A metasearch solution that is emulating user's actions to enter form data has to always start entering data on a freshly downloaded page with an actual value of the parameter.

Ryanair

The *Ryanair* airline offers cheap flight tickets in Europe. Its booking search form is a standard flight booking form that contains three categories: adults, children, i.e. the age is under 16 years, and infants, i.e. the age is under 2 years.

Its search [Rya06] form, shown in Figure 2.9, is a complicated web page with a lot of JavaScript calls to submit form data. Figure 2.9 contains only important parts for an explanation.

Lines 1 and 2 define the search form. Data are sent with the *post* method.

Lines 4 to 7 define the departure airport select. The select tag contains a list of airports defined with the option tags. Every change of selected airport will cause the invoking of a JavaScript method to filter the list of destination

```
1  <form name="Form1" method="post"
2          action="criteria.aspx?lang=en&amp;agencyId=" id="Form1">
3  <input type="hidden" name="__VIEWSTATE"
4          value="dDw2ODY1MjA5Nzk7..."/>
5  ...
6  <select name="_ctl1:cboRetDay" id="_ctl1_cboRetDay">
7          <option value="01">01</option>
8  ...
9  <select name="_ctl1:cboRetMonthYear" id="_ctl1_cboRetMonthYear">
10         <option value="2006/3">March 2006</option>
11 ...
12 <input src="images/next.gif" language="javascript"
13         name="_ctl1:cmdNext" id="_ctl1_cmdNext" type="image"/>
```

Figure 2.8: Interesting parts of the SkyEurope [Sky06a] HTML search form.

airports, i.e. a user can select only valid routes at the beginning of the searching and booking process[1].

Lines 9 to 11 define the departure day select. Similar to the departure airport select, every change causes calling a JavaScript method to check selected dates, e.g. that the selected departure date is before the selected arrival date.

Lines 13 to 14 define the submit button. It does not directly cause a submission of form data, it calls the *submit_SB_Form* JavaScript method specified in the *onClick* attribute instead.

Lines 16 to 17 define hidden parameters that will contain form data that will be processed on the server side. Their values are set in the *submit_SB_Form* method.

Last lines 19 to 20 define the *submit_SB_Form* JavaScript method. The method copies data from the search form to hidden parameters, e.g. line 20 shows how a value of the *date2* parameter is computed, and submits data.

For a metasearch solution this kind of search form is difficult. A design of a solution that sends a URL includes an investigation process to find out how values are transferred between parameters. Any change may lead to a situation where it is necessary to revisit the whole process of value transferral.

A solution that emulates user's actions to enter form data should work without any problems. There can be a complication with changing the content of parameters, such as a dependency between two or more select parameters that requires to filter or limit a content of other select parameters, e.g. the departure - arrival airport dependency. Any dependency change, e.g. a new route or cancelled route, may require a revision.

[1] Several years ago such a feature was not available. A wrong selection was possible, it was checked on the server side and the server produced an error page. This caused unnecessary client - server network communication and prolonged the time necessary to correctly fill out a form.

```
1  <form name="SBform" method="POST"
2          action="http://www.bookryanair.com/skylights.cgi">
3  ...
4  <select name="sector1_o" onChange=
5          "changeOrigDestList(document.SBform.sector1_o);">
6          <option value="aAAR">Aarhus (AAR)
7          <option value="aABZ">Aberdeen (ABZ)
8  ...
9  <select name=sector_1_d onChange='reconcile_dates()'>
10         <option value=01>01
11         <option value=02>02
12 ...
13 <input value="Search for Flights" type="button"
14         onclick="submit_SB_Form()">
15 ...
16 <input type="hidden" name="date1">
17 <input type="hidden" name="date2">
18 ...
19 function submit_SB_Form() {
20         document.SBform.date2.value=""+rYear+rMonth+rDay;
```

Figure 2.9: Interesting parts of the Ryanair [Rya06] HTML search form.

Chapter 3

Form Mapping

A sophisticated metasearch solution creates a universal gateway, called a global interface, that reflects features of searched engines, called local interfaces – in other words, a metasearch solution of car hiring search pages contains different elements as a metasearch solution to flight booking system pages.

This chapter explains a theoretical background of an item mapping from local interfaces to a global interface and vice versa. All interfaces are presented as search forms in our case.

In general there are two form mapping and metasearch solution oriented research fields: general-purpose and special-purpose. This thesis describes a special-purpose oriented metasearch solution.

Firstly a classification of a common search engine will be presented. Then the following section shows an interface representation. Both sections will be used in the third section about a matching of attributes and in the fourth section about creating a general interface. The last section describes a process of the result presentation.

3.1 Search Engines Interfaces Classification

A classification of search engines aims to specify possible different interfaces with characteristic elements.

Based on analysis and classifications mentioned in the [DH97], [MYL02], [Wei04], [CCH03] and [PMHY04] papers, there are the following three types of interfaces:

- Simple Type. Such an interface consists of one search form on one page. For an example see Figures 3.4 and 3.5.
- Divided Type. In this case, a search form can search multiple categories (e.g. books, DVDs). There are the following possible implementations of this category:

- *One form on one page.* An interface allows the choosing of a category, that means it has common elements for all categories and one common way of submission.
- *More forms on one page.* There are multiple visual areas, each with own submission, on one page.
- *More forms on more pages.* Such an interface is divided into more specialised search forms, each on a separate page.

- Wizard Type. A search form is spanned into multiple pages and each requires a submission. In this partial steps a query is built. Only after all pages are submitted, is the query completed and executed, and results are displayed.

  For an example, see Figure 3.6.

The classification described in the [PMHY04] paper is based on analyses of 270 sites and it has 6 types of interfaces. A transformation can be found between this and our model and we can consider our model as a simplification.

3.2 Interface Representation

A search interface is presented as an HTML form. It usually contains elements like *text box, check box, radio button, selection box* and a *submit button.* In most cases each of them has a label, i.e. a descriptive text on the left or the right side of an element.

A user fills out a form, i.e. the user enters values to elements, and submits the form. Data is sent to a server and the server returns a response according to sent data.

The HTML source of a form contains more data that better describe the form, like the *default value* or the *maximum length.*

The difference between a label and a default value of an element is that a label describes the *meaning* of an element and a default value shows one of the possible *contents* of an element.

In the interface representation, an element or a set of elements, mostly visually connected, e.g. in one row, is associated with an attribute; a label is considered as the name of an attribute.

Each element has a format, we distinguish three types:

- Text Type. One-or-multi-row box where a user can enter a free text.
- One Option Type. A set of two or more options, where only one can be chosen.
- More Options Type. One or more options, where none, several or all can be chosen.

Mapping to the HTML tags is straightforward: the text type is a text box or a text area; one option type is a set of radio buttons grouped into one group, i.e. only one can be selected, one check box, or a selection list with the disabled multi-select; and the more options type is a set of check boxes or a selection list

with the enabled multi-select.

Each element has a value type, they are described in detail in the [HMYW04b] paper: *date, time, currency, number,* and *id.* The *id* type indicates that the attribute is used for identification purposes as an order number.

The name of an element, its label (if it exists) and also the default value or the maximal length of an element can help to detect the value type.

There is a relation between multiple elements of an attribute. There are four relationship types, as described in the [HMYW04b] paper:

- Range Type. Two or more elements of an attribute are used to define a range. A very common usage is a price range. For example see Figure 3.1 or Figure 3.3.
- Part Type. It refers to the part of a relationship, like a relationship *author* has two elements *first name* and *last name.* For example, see Figure 3.6, the *Postcode* part.
- Group Type. In this type of a relationship a group of check boxes or radio buttons is used to form a single attribute. The labels of the check boxes or radio buttons are the values of the attribute. For example, see Figure 3.6, the *select a category* part.
- Constrained Type. An element is a constraint for another element - a *parent* element. This element is meaningless without being related to the *parent* element. For example, see Figure 3.6, the *Make* and *Model* parts.

Each attribute A is represented as a n-tuple $A = (N, FT, VT, DV, ML, RT)$, where N is the name, i.e. label, FT is the format type, VT is the value type, DV is the default value, ML is the maximal length and RT is the relationship type between elements of the attribute.

The default value can be one value[1], a set of values[2] or not specified.

The interface representation will be used to describe a match between different search engine interfaces.

3.3 Matching

This section shows different approaches to finding matching attributes and attribute values from various search engine interfaces. Some of the mentioned approaches are described in the [HMYW04b] paper.

The first section explains the semantic matching as the main approach. The second section shows the weight-based matching as a secondary and helper approach.

[1] E.g. for a text box.
[2] E.g. for a set of radio buttons.

3.3.1 Semantic Matching

Semantic relationships between textual forms of an attribute representation point out matching attributes of various interfaces.

The following semantic relationships can be found between a name (i.e. a unique identifier of an attribute, in general a meaningful text that describes a meaning of the attribute), or a default value of attributes. Each of them has a weak and a stronger form – the strong form of a semantic relationship is applicable in different domains, the weak form is applicable only in a specific domain:

- Strong Synonymy. Term t_1 is a synonym of term t_2 if t_1 is an element of a set of synonyms of t_2. Strong synonymy is denoted by $S_s(t_1, t_2)$. For example, $S(pillow, cushion)$.
- Weak Synonymy. It is denoted by $S_w(t_1, t_2)$.
- Strong Hypernymy. Term t_1 is a hypernym of term t_2 if t_1 is more generic than t_2. Strong hypernymy is denoted by $H_s(t_1, t_2)$. For example, $H_s(fruit, apple)$.
- Weak Hypernymy. It is denoted by $H_w(t_1, t_2)$.
- Strong Meronymy. Term t_1 is a meronym of term t_2 if t_1 is a part of t_2. It is denoted by $M(t_1, t_2$. For example $M_s(firstname, name)$.
- Weak Meronymy. It is denoted by $M_w(t_1, t_2)$. For example, $M_w(firstname, author)$. This example shows domain-specific knowledge where the first name attribute[3] in one interface describes the same attribute as the artist attribute in another interface of search engines for books or CDs.

So the main difference between weak and strong semantic relationship is that to define a *weak* relationship between two attributes requires a knowledge of a specific domain.

The semantic matching describes relationships in a natural way for a human being, e.g. terms that describe the same feature tend to have synonymical names – this relationship is easily comprehensible and detectable.

Non-human processing requires a rich vocabulary with defined semantic relationships. Or, as with an expert system, described in the [Jac98] and [sPK89] books, a narrow specific expert area can be covered with a relatively smaller vocabulary with a reasonable number of semantic relationships.

3.3.2 Weight-based Matching

The weight-based matching calculates the matching weight between two attributes or values of attributes and then says whether the compared values match or not.

First a helper pre-processing algorithm is explained and then the metrics are listed.

[3]Together with the last name attribute.

The *normalisation*, as the pre-processing phase of attribute names and values helps to reduce mismatches in non-human processing.

The normalisation consists of processes like lower case or upper case, removal non-alphanumeric characters or reduction of the number of spaces:

- Lower Case. It is denoted by $N_{lower}(t)$, e.g.

$$N_{lower}(\text{"Car"}) = \text{"car"}$$

- Only Alphanumeric Characters. It is denoted by $N_{alphanum}(t)$, e.g.

$$N_{alphanum}(\text{"name-01"}) = \text{"name01"}$$

- Reduction Spaces. It is denoted by $N_{trim}(t)$, e.g.

$$N_{trim}(\text{" author name "}) = \text{"author name"}$$

The main difference between *removal of non-alphanumeric characters* and *reduction spaces* normalisations is that the latter does not remove non-alphanumeric characters, but both of them remove spaces. In other words, if special characters like '-' are necessary, the application of *reducing spaces* is appropriate.

The weight between two attributes or value of attributes t and u is computed by the following metrics, as described in the [HMYW04b]:

- Edit Distance. The edit distance metric, $W_{ed}(t, u)$, is defined as the number of changes necessary to apply to transform the first value t to u. If the edit distance of the values is lower then the allowed threshold T_ed, a positive value is returned, otherwise 0.

- Vector Space Similarity. The vector space similarity, $W_{vss}(u, v)$, is based on a tokenisation of values to get a term frequency of each term in each value. The weight is the Cosine similarity of two strings.

$$W_{vss}(u, v) = \frac{\sum_{j=1}^{m} u_j . v_j}{\sqrt{\sum_{j=1}^{m} (u_j)^2 . \sum_{j=1}^{m} (v_j)^2}}$$

  where m is the number of unique terms, u_j is the term frequency of the jth term in value u and similar v_j is the term frequency of the jth term in value v.

- Value Type Match. If the value types defined in the previous section 3.2 are the same for two values, the value type match, $W_{vtm}(u, v)$, returns a positive value, otherwise 0.

- Default Value. Some attributes may have a default value. A metric based on the comparison of default values t and u is the default value metric, $W_{dv}(t, u)$. If the specified default values are the same, a positive value is returned, otherwise 0.

- Boolean Property. Basically a special case for a single check box based on the value types defined in Section 3.2. Such check box usually means a yes or no selection. A metric based on this phenomenon is boolean property metric, $W_{bp}(u,t)$, where u and t are attributes. If both attributes have the boolean property, a positive value is returned, otherwise 0.

The final weight $W(u,t)$ between attributes t and u is the sum of the above mentioned metrics:

$$W(u,t) = W_{ed}(t,u) + W_{vss}(u,v) + W_{vtm}(u,v) + W_{dv}(u,t) + W_{bp}(u,t)$$

A positive value for each metric is determined per a specific domain.

For example, in a flight prices domain, values of departure and arrival attributes can help to identify these attributes. They are specified as possible values[4], i.e. not as a free text, and their content is similar[5]. It follows that a higher positive value for the *edit distance* metric and a lower positive value for the other metrics is the right constellation.

For a book prices domain, where values are mainly entered as a free text, a higher positive value for the *value type match* metric and a lower positive value for the other metrics is a more appropriate configuration.

Attributes and attribute values with the highest weight value are considered as matched.

3.4 Global Interface

One specific search form contains its attributes and attribute values, i.e. a *local interface* contains its *local attributes* and *local attribute values.*

The matching process finds *local attributes* with the same meaning from a set of *local interfaces.*

A set of attributes that emerges from this process is a set of *global attributes*, that is a set of attributes where each of them can be found in every *local interface* through a correlation based on the semantic matching or the weight-based matching, that has the highest weight of selected metrics.

The name of a *global attribute* is found in a process called *merging attribute*, for more details see Section 3.4.1.

A value or values of a *global attribute*, i.e. a *global attribute value*, emerges in the process of *merging attribute values* process described in Section 3.4.2.

Global interface is an interface that contains *global attributes* with their *global attribute values.* The process of creating a *global interface* is described in Section 3.4.3.

[4] In most cases as a drop-down list.

[5] Airport codes and names are standard.

3.4.1 Merging Attribute Names

A global attribute name is important for creating a human-usable global interface.

A global attribute ga emerged from local attributes la_i, where $i \in 1, .., m$ where m is the number of local interfaces.
The set of names of the local attributes is defined as $N = \{n(a_1), n(a_2), .., n(a_m)\}$, where the $n(a)$ function returns the name of the specified attribute a.
The most frequently occurring name is the name of global attribute $n(ga) = n(a_k)$, where k is the first index of the most used name in the N set.

It may happen, the local interfaces are so varied that none of the names is the most used. Then different techniques may be used like the longest name, assuming that the longest text contains the most of human readable information tokens.

3.4.2 Merging Attribute Values

In general, there are three possible ways to merge attribute values from a number of local interfaces:

- Collection. All values from all local interfaces are collected as values of the general attribute. Any duplicates are removed.

  The (dis)advantages of this solution are:

  + just one request is sent per local interface
  - a huge and confusing amount of values may be displayed to a user
  + real values are displayed to a user

- Generalisation. Relationships are found between values from all local interfaces. Appropriate relationships are used to replace more values with one. Of course, any duplicates are removed.

  The (dis)advantages of this solution are:

  + convenient amount of values is displayed to a user
  - more requests are sent per local interface

- Most Used. Only the most used values are considered.

  The (dis)advantages of this solution are:

  + convenient amount of values is displayed to a user
  - more requests may be sent per local interface
  + easy to develop
  - values can be confusing and wrongly distributed

Possible value types are textual or numerical. Each of them requires special handling.

Merging Textual Values

In the first step, all values for a global attribute from local interfaces are clustered into groups. The selection is based on above the described matching metrics such as synonymy and hypernymy match, edit distance and vector space similarity matching.

Values categorised into groups are processed based on one of the three merging approaches.

The third way, *most used* merging, represents a selection of a limited number of the most used terms in each group.

The second way, *generalisation* merging, presents a more complex selection based on the identification of a representative of a group, that is a hypernym of terms in a group or a hypernym of the most used terms.

The first way is straightforward.

A special category is a translation of terms when local search pages are in different languages.

The translation should be performed from the less used languages to the most used language or to the requested language. Basically, the process is not modified, only a translation pre-processing of attribute values is performed.

A mapping of values is stored for the most used and the generalisation merging. It is used in the creation of a result from a global interface.

Merging Numerical Values

Numerical values require pre-processing where:

- if values are presented in more currencies, they are converted by an exchange rate to the most used or other necessary currency,
- if values are in physical units, these units should be converted to have a uniform unit,
- if values are in different scales, they are converted to a unified scale or to a preferred scale.

After the above-mentioned pre-processing all locale attributes numeric values are unified.

A useful differentiation a numeric values into numerical domains is mentioned in the [HMYW04b] paper:

There are two numeric domains:

- Non-range Numeric Domain. Values of non-range local attributes are merged together as values of the global attribute.
- Range Numeric Domain. A range numeric attribute exists in three types of formats:
  - One Selection List. An attribute consists of one selection list.
    E.g. Less than 5€, Less than 10€, Less than 20€, Less than 50€
    Or see Figure 3.1.

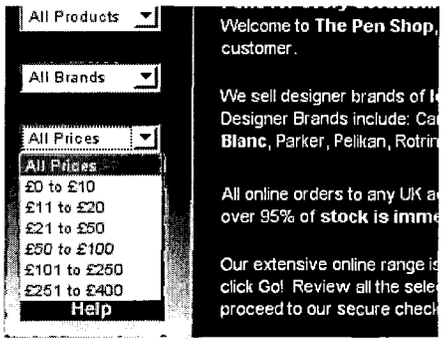


Figure 3.1: The Penshop [Pen06] page shows a selection list for one range numeric attribute for the price range.

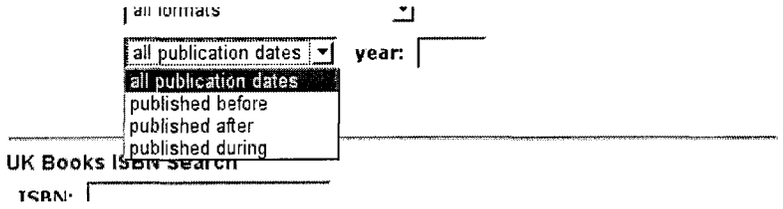


Figure 3.2: The Countrybookshop [Cou06] advanced search page shows a selection list and a text box that specify one range numeric attribute for the publication year.

- One Selection List and One Text Box. An attribute consists of two elements: a selection list for a range modifier and a text box for numeric value.

  E.g. Less, Equal, More ... 100€

  Or see Figure 3.2.

- Two Text Boxes or Two Selection Lists. This type of attribute consists of two elements where each of them can be a text box or a selection list.

  E.g. Year range: after 1968 ... before 1989

  Or see Figure 3.3.

The merging process for the range numeric domain can handle borders specified in a textual form, like:
from a to b means the $< a, b >$ interval,
before a means the $< 0, a)$ interval for a positive number attribute or the $(-\infty, a)$ interval for a number attribute.

The range numeric domain can be merged in two ways:

- to keep the smallest scale steps. This may lead to a high number of intervals in the general interface, but it provides a safe way of merging.
- to use bigger scale steps. This leads to an acceptable number of intervals,

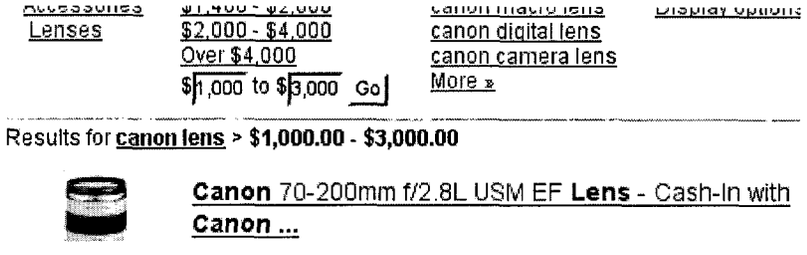


Figure 3.3: The Froogle [Goo06a] page shows two text boxes that specify one range numeric attribute for the price range.

although it can cause that one general request means more requests for a local interface.

For example, the merging of two selection lists from two local interfaces; the smallest scale step is kept:

- 1st:
  - Under $2
  - From $10 to $50
  - From $50 to $100
  - Over $100
- 2nd:
  - Less than €5
  - Less than €10
  - Less than €20
  - Less than €50

it is necessary to convert the first list currency from US$ to €, with the exchange rate US$ 2 for 1€:

- 1st:
  - Under €1
  - From €5 to €25
  - From €25 to €50
  - Over €50

and then the two lists can be unified as values of the general attribute:

- Under €1
- From €1 to €5
- From €5 to €10
- From €10 to €20

- From €20 to €25
- From €25 to €50
- Over €50

Also in the merging of numeric values the general ways of merging can be applied, especially the *collection* way, to use all values from local attributes or the *most used* way to choose only the limited number of the most used values.

Merging Complex Values

The merging of complex values is similar to the merging of a list of values, while values of more parameters are used, not only values of one parameter.

The merging can concatenate parameters or split them. The are no general rules, the complex value merging covers only specific domains. The most used domains are mentioned in the following list:

- Date Domain. A date value can be expressed by one, two or three parameters – the most common configurations are: *day, month, year* or *month & day, year* or *month & day & year.*

  Date domain merging can be denoted in the following ways:

  - one parameter to three parameters is denoted as

    $$M_{date13}(param, format) = \{param_{day}, param_{month}, param_{year}\}$$

    that is a $param$ with the specified format $format$ is split into three parameters day, $month$ and $year$ with their values $param_{day}$, $param_{month}$ and $param_{year}$.
  - three parameters to one parameter is denoted as

    $$M_{date31}(param_{day}, param_{month}, param_{year}, format) = \{param\}$$

  - for two parameters to one or vice versa the denotation is similar.

  The most important is the $format$ input parameter that defines the content of a specified parameter. Possible values of this parameter are defined in this list:

  - d defines the day part of the parameter value.
  - m defines the month part of the parameter value.
  - y defines the year part of the parameter value.
  - a non-numeric value defines a delimiter like '.' or ','.

  Example of a date domain merging:

  $$M_{date13}(param, \text{"m.d.y"}) = \{param_{day}, param_{month}, param_{year}\}$$

  where $param$ = "11.17.1989". The parameters will contain the following values: $param_{day} = 17$, $param_{month} = 11$ and $param_{year} = 1989$.

The merging process should also handle relative notation of a date, such as *Today* or *Tomorrow*, where the textual form is translated into the absolute value and then processed like other absolute dates.

The majority of relative date textual forms depend on the actual date.

- Currency Domain. A currency can be expressed by one or two parameters – the configurations are the following: *number & currency*, *currency & number* or *number, currency.*

The currency domain merging from one parameter to two parameters is denoted as

$$M_{curr12}(param, format) = param_{number}, param_{curr}$$

that is the *param* parameter is split into parameters *number* and *currency*.

Merging from two parameters to one parameter is denoted similarly:

$$M_{curr21}(param_{curr}, param_{number}, format) = param$$

Again the most important input parameter is the $format$ parameter:

- c defines currency of the parameter value.
- n defines number of the parameter value.
- a non-alphanumeric delimiter that delimits the previous two parts. E.g. ' ' (space) or ', ' (comma and space).

Example of a currency merging:

$$M_{curr12}(param, \text{"c n"}) = \{param_{curr}, param_{number}\}$$

where $param$ = "US$ 100". The parameters will contain the following values: $param_{curr}$ = "$US$$" and $param_{number} = 100$.

3.4.3 Creating Global Interface

A global interface contains global attributes with their global attribute values and a global attribute name.

The usability of a global interface that contains all global attributes can be reduced if the number off all global attributes is too high, e.g. over 10 global attributes. There is a possibility to reduce their number, i.e. only the selected attributes will be available for an end user.

The process of the reducing of the number of all global attributes can be based on the number of local interfaces where a mapping for the processed global attribute exists, i.e. a higher occurrence of a global attribute indicates a more *important attribute.*

Although a global interface can be created automatically, it should only represent the starting point of the creating process, e.g. where the order of attributes corresponds to their *importance*, but the final layout has to have a human blessing.

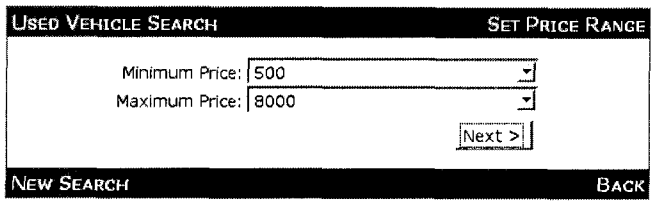


Figure 3.4: The first search form of the Autolocate search engine.

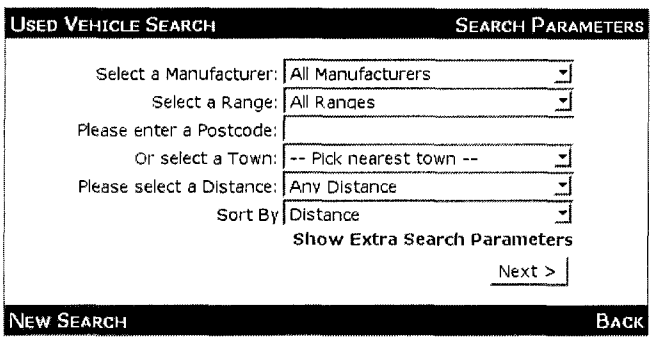


Figure 3.5: The second search form of the Autolocate search engine.

Example

In this example, a global interface will be created from two search forms.

The first search engine *Autolocate* on [Aut06] page, has the wizard type of interface. It consists of three steps – search forms. *Only the first two of them will be shown for simplicity in this example.* The search forms are shown in Figures 1.4 and 1.5.

The second search engine *Carland* on the [Car06] page, has the simple type of interface. The search form is shown in Figure 1.6.

The process of creating the general interface for the above-mentioned search engines is the following:

1. Let's assume that the requirements are to search for a price and a manufacturer.

2. No conversion of values is necessary – both of search engines search in the

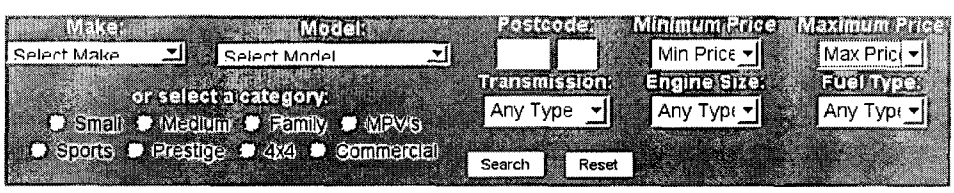


Figure 3.6: The search form of the Carland search engine.

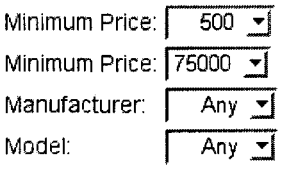


Figure 3.7: The global interface search form for the two car search engines.

UK.

3. The attributes *Minimum Price* and *Maximum Price* are in both search forms and therefore they will be added to the general interface.

   The values will be created by the numeric value merging.

4. Matching of the *Manufacturer* and *Make* attributes is more complicated and can be done in two ways:

   - By comparing values of the attributes. They contain very similar data; the edit distance metric has the highest value for these two parameters.
   - By the strong synonymy and the strong hypernymy – *manufacturer* and *maker* are synonyms; *maker* and *make* are hypernyms.

   The values of the general attribute *Manufacturer* (because the name is longer and may contain more information than *Make*, although in this case the informational value is the same) will be created by the textual values merging process.

5. The local attributes *Range* and *Model* can only be considered as matching when using the following ways:

   - By comparing values of the attributes.
   - Domain knowledge that range and model mean the same, i.e. the weak synonymy semantic relationship.

   The name of the general attribute will be *Model*. The values will be created by the textual value merging process.

6. The graphical layout is done manually and it is shown in Figure 3.7.

3.5 Global Result Presentation

The intention of this section is not to describe in detail such a complicated topic as the global result presentation is, solely to provide the whole picture of a meta search solution that also needs to display obtained results.

A global interface is used as a unified interface for all local interfaces.
A user enters data in a global interface, then a system is able to translate global attributes and their values to local attributes and values and to perform

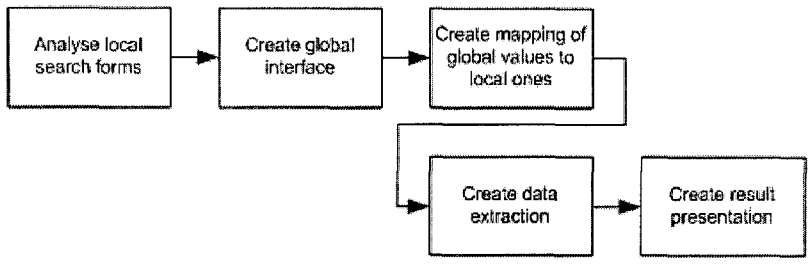


Figure 3.8: An overview diagram of creating a metasearch scenario.

requests to search engines presented by local interfaces.

Particular results obtained from search engines need to be presented in a unified form. The process of creating a *global result presentation* is similar to the process of creating a *global interface*, but the technical background is different.

A global interface is created from a relatively clearer environment of attribute – value pairs, that are technically distinguished with special tags; whereas a global result presentation does not have a friendly technical environment – resulting data can be presented in various forms such as simple text, table, a set of rectangular areas[6], basically in an unstructured and unorganised way.

A solution to extracting data from results is to use a wrapper such as *Lixto Visual Wrapper*. For more details, see Section 4.3.2.

There are techniques to merge search results automatically, as described in the [CHT99] and [LMS+05] papers. Another automatic search result merge approach, described in the [ZMW+05] paper, uses graphical information like position of a block of data to distinguish relevant and irrelevant data.

The final presentation should be the result of human-guided process that is more appropriate, less error-prone and results in a easy to understand, structured and well-arranged format.

The process of creating a metasearch scenario is summed-up in Figure 3.8.

The mapping from local interfaces to a global interface can be applied in an automatic creating of a global interface, e.g. a wizard in an application for creating scenarios. The mapping from a global interface to local interfaces is used in metasearch solutions for processing requests, i.e. to map values from a global interface to particular local interfaces – search engines.

[6]The `div` tag.

Chapter 4

Existing Solutions and State of the Art

This chapter describes existing solutions and products that will be used for the design and implementation of the Snorri metasearch solution.

The first section shows existing meta search engines, approaches and papers.

The following sections describe the starting point for new requirements and design – the existing *Lixto MetaSearch* product with other related products included in the *Lixto Suite*; designed, researched by the DBAI at the Vienna University of Technology and developed by Lixto company.
The first part describes the main products of the Lixto Suite and the second part describes the current Lixto MetaSearch version.

4.1 Existing Meta Search Engines

Existing meta search engines can be split into academic and commercial engines. The first group focuses on and solves particular problems and most of the solutions do not concentrate on a high use load. The second group needs to create very effective solutions and avoid slower responses for an end user.

4.1.1 Academic Projects

Most of academic projects focuses on the theoretical background of a metasearch engine and related topics.

The first papers in this area were focused on the analysis that existing search engines cover a very small range of the pages on Internet, and that a combination of several search engines provides better results.
The [LG98] paper explains that a single engine does not index more than about one-third of the *indexable Web* and says: *"The coverage of the six engines investigated varies by order of magnitude, and combining the results of the six engines yields about 3.5 times as many documents on average as compared with*

the results from only one engine."

A lot of meta search engines were created at universities: Inquirus, MetaCrawler, SavvySearch, ProFusion and meta search for images MetaSEEk. Some of them operate as commercial services.

For more details see below.

One recent group from the University of Illinois focused on the creation of an automatic meta search engine that is described in their papers [WMYL01], [HMYW04a], [PMHY04], [HMYW04b], [WYDM04] and [MWR+05]. Their recent papers [LMS+05] and [ZMW+05] focus on automatic result merging. They created a system called *WISE*. Its features are mentioned below, in the last section.

In the [WYDM04] paper they mentioned that they are investigating: *"the possibility of user interactions in resolving other uncertainties in the matching process"*.

The [HZC05] paper describes a tool for extracting and matching web query interfaces – MataQuerier. The paper says: *"The MetaQuerier system fully automates all the tasks in streamline to output semantic matchings"*. For demonstration purposes the tool can display matching elements.

The [HC03] paper describes a new approach based on a statistical schema matching. The paper says: *"Unlike traditional approaches using pairwise-attribute correspondence, given a set of input sources as observed schemas, we will find hidden models that are consistent, in a statistical sense, with the schemas observed. Using a scenario of matching"*.

There is also ongoing metasearch research at the University of Freiburg. They use a fully automatic wrapper tool to extract the results received from search engines. For more details see the [Sim06] page.

In general, research has moved from simple search form engines to more complicated search form engines with automatic mapping of a search form to the automatic processing of results.

Inquirus

The Inquirus metasearch engine described in papers [GLG+99], [GLBG99] and [Glo01] was created to avoid difficulty in merging the results from multiple search engines. The [GLG+99] paper says: *"Inquirus downloads and analyzes all pages listed by the search engines. With the full-text of all pages, the document ordering problem returns to the easier, but still very hard, problem encountered by standard search engines."* The architecture allows the ability to display query-sensitive summaries, results that are always up to date with the current contents of the Web (improving relevance), and improved duplicate detection.

The Inquirus meta search engine also allows searching for images. This functionality also facilitates the MetaSEEk meta search engine, for more information see below, but MetaSEEk targets query by example as opposed to keyword search, which is the focus of the image metasearch functions of Inquirus as described in the [LG99] paper.

MetaCrawler

The MetaCrawler metasearch engine provides a single, central interface for Web document searching. It receives a query and posts the query to multiple search services in parallel, collates the returned references, and loads those references to verify their existence and to ensure that they contain relevant information. The [SE95] paper says: *"The MetaCrawler is sufficiently lightweight to reside on a user's machine, which facilitates customization, privacy, sophisticated filtering of references, and more."*

It is available on-line on the [Met06] site as a commercial service.

SavvySearch

The [HD97] paper says: *"SavvySearch is designed to balance two potentially conflicting goals: maximizing the likelihood of returning good links and minimizing computational and Web resource consumption. The key to compromise is knowing which search engines to contact for specific queries at particular times."* The engine tracks long term performance of search engines on specific query terms to decide which are appropriate. It also decides if to contact a search engine at all by monitoring recent performance of the search engine.

It was absorbed by Search.com and it is available on-line on the [Sea06] site as a commercial service.

ProFusion

The ProFusion metasearch system, described in the [Gau97] paper, is a Web meta-search engine that supports automatic query dispatches. It analyzes incoming queries, categorizes them, and automatically picks the best search engines for the query based on a priori knowledge – *confidence factors* – which, as described in the [Gau97] paper, *"represents the suitability of each search engine for each category. It uses these confidence factors to merge the search results into a re-weight list of the returned documents, removes duplicates and, optionally, broken links and presents the final rank-ordered list to the user"*. The [BBC97] paper says: *"ProFusion's performance has been compared to the individual search engines and other meta searchers, demonstrating its ability to retrieve more relevant information and present fewer duplicate pages."*

ProFusion meta search is also available on-line on the [Pro06] site as a commercial service.

MetaSEEk

MetaSEEk is an integrated search engine, which serves as a common gateway, linking users to multiple image search engines. The [BBC97] paper says: *"It includes three main components: display interface (for a client), meta search engine and query interface (for search engines)."* The query interface component receives search queries from a user and translates them to the specific query interfaces used by each target search engine. Then the dispatching component

decides which search engines the query should be sent to. The last component, the display component merges the results and ranks them for display.

The [BBC98] paper says: *"It evaluates the performance of each query method on a search engine for future queries based on the user's feedback."* This information is then used to optimise and modify the corresponding entries in a performance database.

The feedback page is shown in Figure A.5 on page 148.

WISE

The [PMHY04] paper describes a new approach of clustering e-commerce search engines so that clustered engines in one cluster sell similar products. This fact allows the building of a metasearch engine over all e-commerce search engines in a cluster. The [PMHY04] paper says: *"Our approach performs the clustering based on the features available on the interface page (i.e. the Web page containing the search form or interface) of each ESE. Special features that are utilized include the number of links, the number of images, terms appearing in the search form, and normalized price terms. Our experimental results based on nearly 300 ESEs indicate that this approach can achieve good results."*

For more details see a list of related papers and projects on the [Men06] page.

4.1.2 Commercial Projects

The commercial metasearch engines can be split into the following categories: handling of general search engines like Google, handling specialised search engines and complex metasearch solutions.

General Search Engines

Search engines like Google, Altavista or Yahoo have and are focusing to have very simple user interfaces, i.e. search forms.

To create a mapping for those search engines is very simple, there is practically just one parameter – the search phrase.

Although a search phrase has a simple syntax, like using '*+*' or '*element1 element2*', practically all of the above-mentioned search engines have the same syntax, as described on their help pages [Goo06b] and [Yah06].

A metasearch engine basically has a simple job:

1. map one global parameter to several search engines with one parameter
2. perform queries to search engines
3. retrieve results
4. transform results like sorting, filtering and removing duplicates
5. display results

The main difference between metasearch engines with low complexity of searched engines is in the transformation and presentation of results. All of them cover the well know search engines.

The *Mamma* metasearch engine, available on the [Mam06b] site, uses for transformation its own *rSort* algorithm – rSort works like a voting system for search results. Their [Mam06a] about page says: *"The search engines Mamma.com queries often return duplicate results. Instead of simply eliminating the duplicates as many metasearch engines may do, we use this information to rank our results. Each duplicate search result is considered a 'vote' for that result."*

The *KartOO* metasearch engine, available on the [Kar06] site, presents its results in an interesting graphical form – maps. For example see Figure A.6 on page 149.

The *jux2* metasearch engine, available on the [Jux06] site, facilitates the comparing of results between Google, Yahoo and MSN search engines, that is it can show what e.g. Google found and what others did not and vice versa.

The *Vivísimo* metasearch engine, available on the [Viv06b] site, facilitates clustering results. For example see Figure A.7 on page 149.

Specialised Engines

A more complicated search form indicates a more specific search engine. A good example are search engines for shops or the travel industry, like flights, hotels or car rental.

The flight metasearch engine *Check Felix*, available on the [Che06b] site, searches more then 170 particular airline and travel agent pages that are described on its [Che06a] page.

Its situation is completely different compared to general search engines – it has to map nine parameters, see its [Che06b] main page, to various pages from different countries with unstable parameter names.

The travel metasearch engine *Kayak*, available on the [Kay06b] site, describes its technology on the [Kay06c] page: it searches 100 on-line travel sites, hundreds of airlines, thousands of hotels and leading rental car brands.

The [Kay06a] specification for a hotel search partner defines that the partner has to provide a CSV or an XML file.

The flight metasearch engine *Skyscanner*, available on the [sky06b] site, searches for cheap flights in Europe and Australasia. Their specific approach is based on the fact that users start the update process for the price of the selected flight if the price was not recently updated. The engine essentially only updates flight prices chosen by users and it does not update all flight prices automatically. The most demanded flights are collectively updated in a short time.

From the technological point of view the above-mentioned metasearch engines, like others in this group, have hardwired mappings for particular search engines.

Complex Solutions

Some companies provide products for creating metasearch like solutions.

The *Fetch* company provides the *Fetch Agent Platform* for extracting and integrating information from multiple Web sources, and the [Fet06] page says: *"to transform the data into a form that is useful for business applications."*

The *Fetch Agent Platform* is composed of AgentBuilder, the design-time component, and AgentRunner, the run-time component. Additional capabilities, including the ability to extract named entities and to normalize terminology across sources, can be deployed as part of an AgentRunner installation.

They also did research in the area of automatic wrapper maintenance that is described in the [LMK02] paper.

The *Vivísimo* company, see the [Viv06b] company site, provides the *Velocity* platform for crawling and indexing documents and databases, searching all sources with just one query and it offers, as mentioned on the [Viv06c] page, search results organized in folders rather than long lists.

A part of the *Velocity* platform is the *Content Integrator* product that facilitates search query through a single search interface that acts as an intermediary to various informational repositories. The query is sent simultaneously to all designated search sources and results are returned to the user in a single list.

The *Content Integrator* [Viv06a] specification says that the company provides handlers for the most commonly used search engines (Google, Autonomy, FAST, Ultraseek, Verity, etc.). A custom search engine, that is not included, requires a minor customization to complete the integration process.

4.1.3 Lixto Project

The Lixto Suite presented below with the Lixto MetaSearch product is comparable to common metasearch commercial solutions.

The Lixto Suite facilitates a strong platform for accessing Internet sources with all features like HTTPS and Java Script. The Lixto MetaSearch is a product to write hardwired metasearch solutions.

4.2 Lixto Suite

The Lixto Suite consists of two products: *Lixto Visual Wrapper* and *Lixto Transformation Server.*

Each of them solves a different area of problems and the connection between them is a XML document as an universal exchange format.

The Visual Wrapper part provides downloading and extracting an XML document from an Internet source, like a web page.

The Transformation Server part provides data flow processes like collecting, transforming, concatenating, sending and storing for XML documents.

Together they present a complete solution to data processing from the Internet.

MetaSearch uses the first product, Visual Wrapper, that provides structuring capabilities for Internet sources.

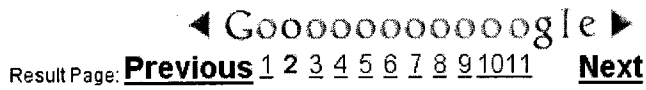


Figure 4.1: Result pages navigator on Google. (Probably the most famous pages navigator in the world.

4.3 Lixto Visual Wrapper

The Lixto *Visual Wrapper*, *VW*, that is described in the [GKB+04], [BFG01b] and [BFG01a] papers, is a product that provides a visual and an interactive way to create a set of instructions, called *wrapper*, for extracting of data from web pages into an XML document.

The product consists of modules such as Navigator, Extractor and Visual Developer that are described in the following sections.

4.3.1 Navigator

The navigator module is able to load a start page, defined by an URL and then mimic actions of a user like moving and clicking with a mouse or typing on the loaded page.

User's actions cause different changes that depend to a greater extent on the defined functionality of the loaded page and to a lesser extent on the standard browser functionality.

The recording of user actions is more powerful than a recording of request – response behaviour because it is able to record the behaviour of internal functions of a page, e.g. a page can change its structure by clicking on a link or button – this is a functionality of a page; or a browser creates the `onSubmit` event when a user clicks on the submit button and the related form data are sent to the specified URL – this is the standard functionality of a browser, although this behaviour can be changed by the internal functions of a page, for example validation of data.

One important change is a change of the actual URL, where a new page is loaded and the whole process continues on the fresh loaded page.

This allows a navigation through several pages, i.e. portals with a login screen, pages with a menu or *wizard* style pages, that is a requested result page is retrieved after several query pages.

Also *search engines* style pages are covered, which is very important for MetaSearch, i.e. pages, where the required result is spread over more pages that are accessible through a page navigator, as in Figure 4.1.

The Navigator process consists of the following steps, for a diagram see Figure 4.2:

1. Receive a request.
2. Start a navigation sequence with the starting URL.

3. Send recorded user actions like moving with the mouse, keystrokes, clicks. These actions may lead to change of URL or page structure. All actions can be parametrised.

4. Last recorded actions are sent; the last page is returned and it is used in the Visual Wrapper module.

The next step is to extract data from a loaded page.

Parametrisation

A loaded page is a result based on user's requests that are recorded in a navigation. A navigation can be parametrised, that is a process using this navigation can specify the values of parameters.

Specific recorded user actions can be marked as a parameter like entering a name of a product. The replaying of a recorded navigation uses values of parameters, i.e. it replaces recorded values with parameter values.

This allows widely reuse of a created navigation for e.g. searching for products with different attributes.

A very important and complex topic in this area is the mapping between parameters of different search engine forms, i.e. values of parameters are defined on a general level and are used for different search forms with different parameter features like name and value metric.

For more information see Section 5.7.

4.3.2 Visual Wrapper

The Visual Wrapper or extractor module mines requested data from a page into a predefined output structure.

Because the result is an XML document, the predefined output structure is given by a DTD or xSchema description.

The Visual Wrapper process consists of the following steps, for a diagram see Figure 4.2:

5. Requested data are selected by the *Elog* extraction language or XPath.

6. Selected data are mapped to the output structure.

7. The resulting XML document is created and returned.

Wrapping Search Forms

A wrapping process, in general, allows the extracting of data from a web page. For most cases, a result is based on human readable information from the extracted web page.

But the wrapping of search forms can also provide machine readable information like parameter names and values of hidden parameters.

Obtained information can be used in two processes:

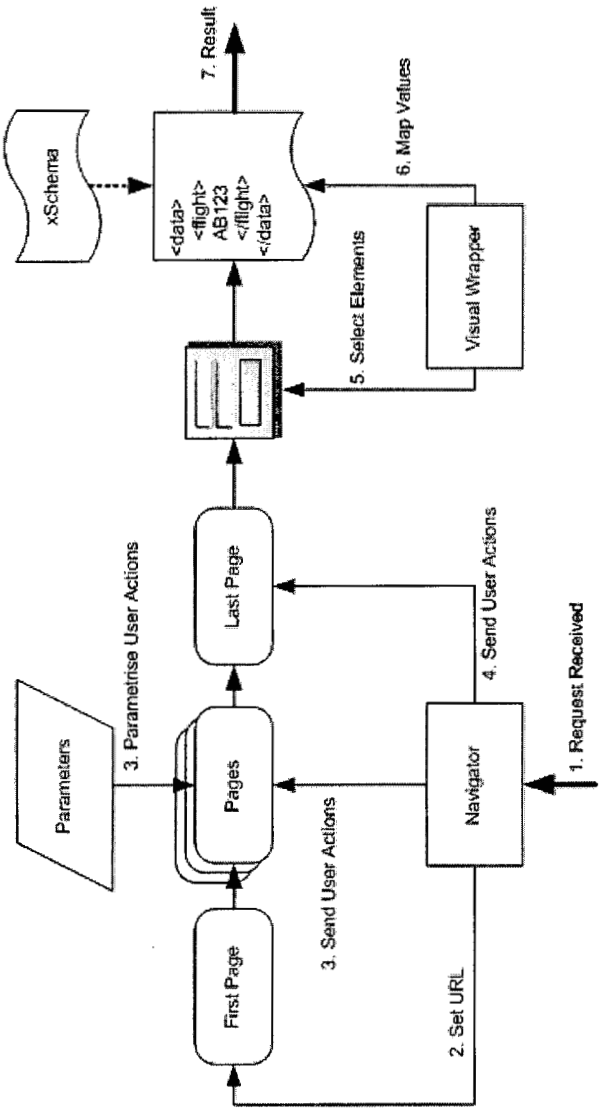


Figure 4.2: The processing of Navigator and Visual Wrapper. Navigator sends the first URL and then applies recorded user actions to the page. The last page is then processed by Visual Wrapper, where elements are found and their values are extracted to a XML document.

- A mapping creating. A graphical interface or a helper tool can use this data and speedup the process of creating a mapping like to display a list of all parameters or to find automatically matching parameters and values.
- The URL mapping, as shown in Section 5.6. This feature allows to dynamically and automatically find matching parameters and values, i.e. a developer does not have to fully specify a mapping; its vague formulation can describe relationships between elements that leads to a non-fixed mapping.

4.3.3 Visual Developer

The *Visual Designer* product is a visual tool for creating a complete navigation and extractor procedure, that is also called a wrapping.

The user graphical interface is inherited from the Eclipse interface, that is represented as a mature interface and world leaders in development tools have claimed they will use Eclipse for creating their own tools.
A user uses a full-featured and world-wide accepted Mozilla browser integrated into the user interface for creating a navigation and an extraction.

An example of the wrapping mode with highlighted anchors on a page, see Figure A.3 on page 146.

4.3.4 Usage of Mozilla

Mozilla is a mature browser based on Netscape Navigator fundamentals.

The browser engine, Gecko, was designed to support open Internet standards such as HTML 4.0, CSS 1/2, the W3C Document Object Model, XML 1.0, RDF, and JavaScript, that includes also AJAX, as described on the [Moz00] page.

But Mozilla is not only a browser, it is also a cross-platform application framework, a base for many other browsers and last but not least an open source community, see the [Moz05] page for details.

4.4 Transformation Server

Lixto Tranformation Server, *TS*, described in the [Her02] thesis, is a product that in general provides data flow between various components.
The TS solution is a complex system that has large input and output possibilities, with rich internal functionality. Almost all features are accessible through a graphical user interface. It defines several user roles that have specialised user interfaces.

The following sections describe TS features.

MetaSearch is basically a subsystem of the TS system. Although they were developed separately, they share the same ideas and solutions for common problems.

The new MetaSearch version as described and prototyped in this thesis brings J2EE technologies like EJB that are used on a limited area compared to TS. It is easier to solve obstacles and pitfalls that will emerge in a smaller project and then predict more accurately possible problems and go around already know pitfalls in a larger project like TS.

4.4.1 Components

Each component has its functionality like transformation or delivering data to an external system. All components have specified allowed inputs and outputs; the number of them is limited by the specification. The transmission medium between components is an XML document.

Almost all component types have the possibility to be started at defined times. This functionality is called *scheduler*.

A schematic picture of the processing is shown in Figure 4.4.

A set of components is called a process or a scenario or a pipe. It groups components to achieve a functionality like to read data from required sources, integrate it and send by an e-mail to a user. A pipe is a directed acyclic graph.

When a component is started, it applies its function to its inputs. When inputs changed, the result is also changed and this change is propagated to the following components, i.e. they are started by this change.

In TS there are several components, the main ones, from the metasearch point of view, are mentioned in the following list:

- Source. The Source component facilitates obtaining data from various sources as Internet, by means of VW, database or web service. It is the first component in a pipe.
- XSLT. The XSLT component facilitates a transformation of an input XML document with an XSTL. The Transformer and Integrator components facilitate specialised but simplified functionality of the XSLT component.
- Composer. It provides a specialised transformation of an input XML into formats like simple text, PDF, picture. Basically every deliverer component has its composer component.
- Deliverer. It facilitates delivering data to various systems like e-mail, FTP, database. It is the last component in a pipe.

4.4.2 User Roles

As mentioned above, TS is a complex system. It facilitates its functionality from more points of view, i.e. several user roles were defined and each of them has its user interface optimised for the user role tasks.

The following user roles and user interfaces were defined:

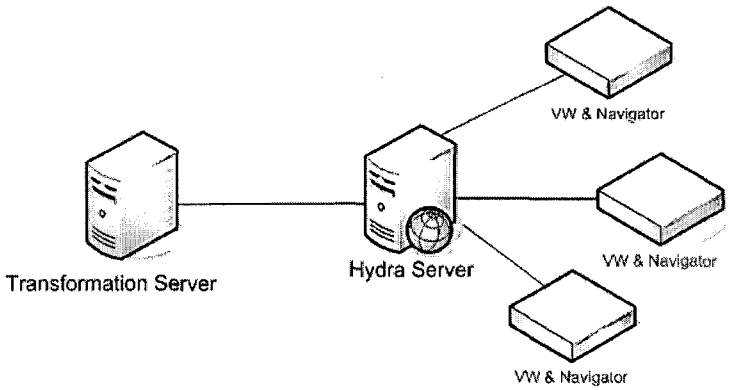


Figure 4.3: Hydra and its position between Transformation Server and Navigator and Visual Wrapper.

- Administrator. It facilitates the server configuration and it allows to monitor and manage running pipes.
- Scenario Designer. A workbench for a scenario developer. It allows to create new scenarios, add components and configure them.

  For a screenshot see Figure A.4 on page 147.
- User. A lightweight user role with simple interface facilitates the using of created pipes by a developer.
- Manager. A role between a developer and a user. It allows to manage a limited set of pipes used by users.

4.4.3 Hydra

Hydra is a layer between Transformation Server on the one side and Navigator and Visual Wrapper on the other hand, see Figure 4.3.

Generally, it provides an unified interface between an application and couple Navigator and Visual Wrapper.

Hydra allows Transformation Server or any application in a simple way to use all the power of Navigator and Visual Wrapper.

There are three main advantages of Hydra:

- It allows to separate and install Hydra with Navigator and Visual Wrapper on one server and on another application like Transformation Server on another server.
- It hides any possible problems with Navigator or Visual Wrapper. If there is a problem with a not responding Visual Wrapper instance, the process can be terminated and start a new instance, that will start again or continue in a process.

- The last advantage is technology-dependent. A Java enterprise application that uses EJB and runs on an application server can not use the JNI library, for more details see the Sun EJB specifications [Sun03b] and [Sun05b]. Navigator and Visual Wrapper use the JNI library to communicate with Mozilla.

Hydra is a multi-threaded Java application that publishes an interface to applications and uses the Navigator and Visual Wrapper common interface to communicate with them.

Hydra starts instances of Navigator and Visual Wrapper locally on a server, i.e. typically on a server there is one Hydra instance and many Navigator and Visual Wrapper instances started by the Hydra instance.

4.5 Lixto MetaSearch

This section is dedicated to the actual version of Lixto MetaSearch, that is described in the [Ros04] thesis.

The first sections describe requirements, the second the state of the art. The third and the following sections show problems of the current version. The last section lists new requirements for the new version.

4.5.1 Requirements

Transformation Server is a heavyweight solution with complex data flow processing. Its main principle is based on processing at defined times, where a result is delivered to many users.

MetaSearch complies with completely different requirements.

It was created to provide a highly optimised environment able to process multiple requests in undefined times, where each request required its processing.

In other words, MetaSearch has to be fast but reliable.

See Figure 4.4 and Figure 4.5 for a schematic picture of the differences between the Transformation Server and the MetaSearch.

4.5.2 Current Status

The current MetaSearch implementation consists of three main parts:

- Engine. The multi-threaded core of the system.

  It processes a received request: creates a *job* that is put to a pool of jobs; a main system thread takes a job from the pool and executes it; in the last step all results (for more search engines) are taken and transformed.

  The execution of a job consists of the following steps; see Figure 4.6 for a work flow diagram:

  1. For each job a request for the corresponding search engine is prepared. This part is sequential.

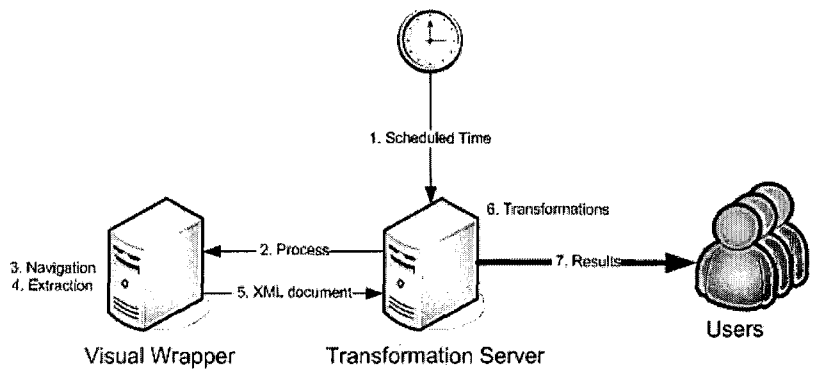


Figure 4.4: A schematic picture of the processing in Transformation Server. Results are generated in scheduled times and sent to users. There is heavy traffic only in the last step, sending results to users.

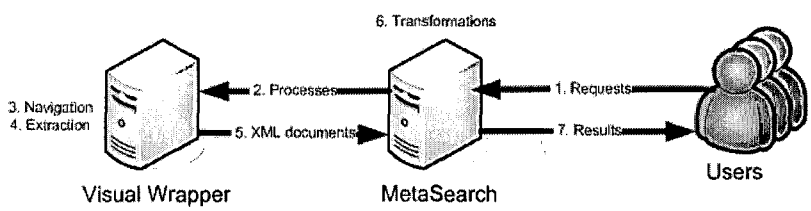


Figure 4.5: A schematic picture of the processing in MetaSearch. Results are generated on a request by users and sent back to users. Heavy traffic is basically in the whole system.

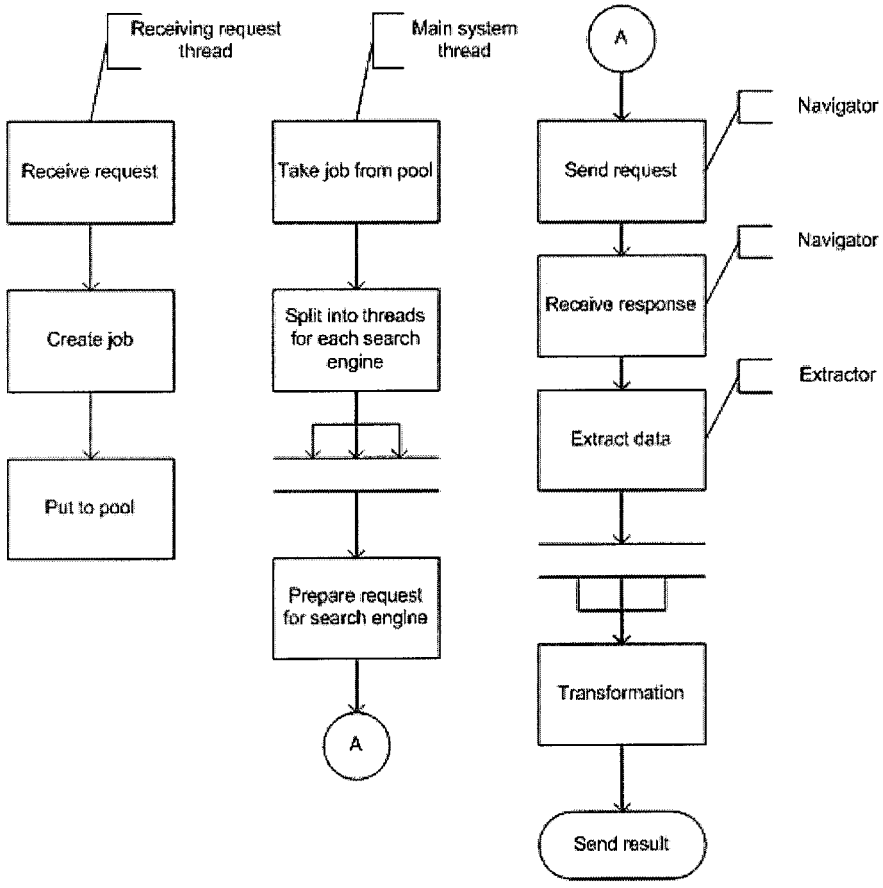


Figure 4.6: The work flow of the current MetaSearch version. The last but one process *Transformation* is one of the problematic points because the MetaSearch is not able to process it parallel.

This point is problematic for complicated scenarios that require a longer time for preparing a request. The problem was partially solved by creating the possibility to process a part of the preparation of a request parallel; it is a solution that broke the design.

2. A job is split into threads, for each search engine one. This allows a concurrent processing and faster delivery of result.
3. A request is sent, a response is downloaded and the result data are extracted via an instance of Visual Wrapper.
4. When all the results are received, the final transformation of results is performed.

   This is one of the problematic points because it is not in the parallel part of the processing.
5. The final result is ready.

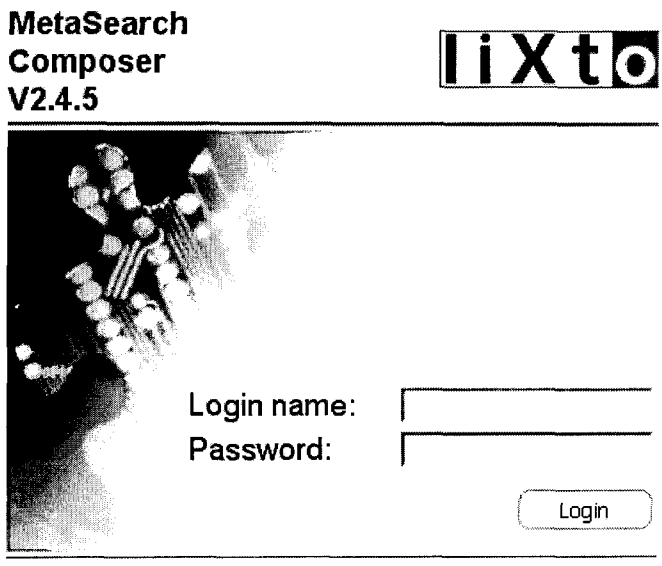


Figure 4.7: The login screen of the current MetaSearch version.

- Web Layer. The web layer is implemented in WebWork, for details see the [Ope06] page, a MVC, defined in the [Sun06c] specification, is a framework similar to Apache Struts, that is described on the [Apa06b] page, but simpler.

  It receives all HTTP requests, they are then dispatched to the engine part or to the administrator graphical user interface.

- Administrator Console. A user web interface that allows to manage installed search engines handlers, to enable or disable them, or to configure them.

  See the 4.7 figure for the login screen and the 4.8 figure for the list of installed handlers.

The engine part is the oldest, the last two were added later with a medium impact to the engine part.

There are settings in the configuration of MetaSearch that limit various numbers and give protection from problems such as not enough memory, high processor load, high network traffic:

- the maximum number of concurrently received requests. A request beyond the limit is refused.
- the size of the downloading processes pool
- the size of the extracting processes pool

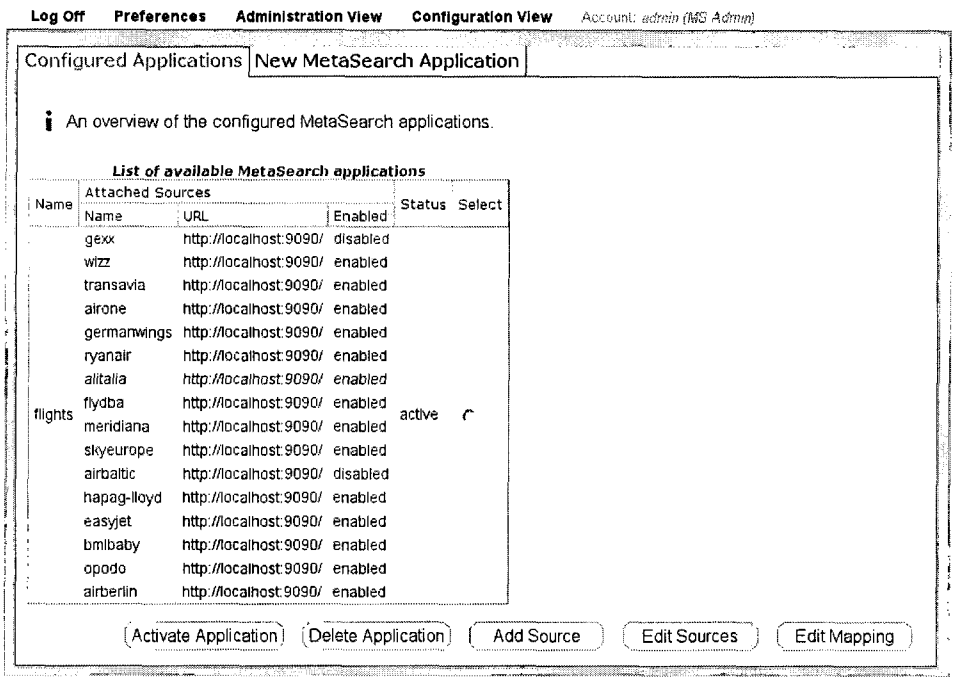


Figure 4.8: The administrator login screen of the current MetaSearch version. The list of handlers is in the *Name* column of the *Attached Sources* header. The buttons on the bottom provide management of the installed handlers.

```
1 <source name="truckscout">
2   <parameter name="manufacturer">
3     <value mappingvalue="120" originalvalue="ACCORD"/>
4     <value mappingvalue="123" originalvalue="AEBI"/>
5 ...
```

Figure 4.9: An abstraction of values of the local parameter *manufacturer* for the *truckscout* source.

A pool has the following features:

- processes can run concurrently
- to limit memory and processor needs
- each process is created and stored in a pool to reduce the startup time or initialising of a process
- each process is reusable, that is when the task of a process is finished, this process can work on a next task without any influence from the previous task(s)

Processes in the downloading and extracting pools are shared between all executed jobs.

The current version is maintained, that means small features and bug fixing, and is used in several projects.

4.5.3 Mapping Framework

The mapping framework provides mapping from the global interface to local interfaces.
Its functionality is divided into two parts:

- Mapping. It covers mapping of parameter values.
- URI Builder. This part provides mapping of parameters.

Mapping

In the mapping every local source has mapping file, that makes an abstraction of local values of parameters; for example see Figure 4.9.
An abstraction of local values can be done manually or automatically. Visual Wrapper is used for the automatic generation, where a page with a form is loaded by it, then all form parameters are extracted and stored to a mapping file.

All abstracted values are mapped in a global file, where each global value is mapped to different local sources, to corresponding abstraction of local values.
For example see Figure 4.10.

The mapping between global and abstracted local parameters is divided into three categories, where each category covers special cases:

```
<parameter name="type">
  <manualmapping>
    <value originalvalue="5.4">
      <source name="tec24">
        <mappingvalue>Pflanzenschutzspritzen</mappingvalue>
      </source>
      <source name="truckscout">
        <mappingvalue>Pflanzenschutz</mappingvalue>
      </source>
      <source name="case">
        <mappingvalue>Used sprayer</mappingvalue>
      </source>
    </value>
...
```

Figure 4.10: A manual mapping for the *5.4* global value of the *type* parameter. The targets are local sources *tec24*, *truckscout* and *case* with abstracted local values.

- Direct Mapping. This covers trivial cases; the value of a global parameter is mapped to the value of a local parameter. It is used mainly for text and hidden fields.

  Basically it is the $1:1$ mappings.

- Domain Mapping. The domain mapping is used for a parameter with a finite set of values. It is used for selects, radio-buttons and check-boxes.

  A value set of a global parameter is mapped to values of local parameters, where each value of a global parameter can be mapped to a set of values. This leads to a situation, where the corresponding local source has to send more requests, for each request at least one.

  It is the $n:m$ mapping.

- Custom Mapping. The last type of mapping in the current MetaSearch version provides support for $1:n$ mappings and for special $1:1$ mappings.

  A scripting language can be used in mappings to cover special implementation needs, like transforming date values. In other words it provides a flexible way to comply with various needs.

An abstraction of local values of a local source and a global mapping file present a complete mapping of values form a global interface to local interfaces.

URI Builder

The URI building process maps global parameters to local parameters for each local source by generating HTTP GET or POST requests.

For each source a separate URI is generated, or in special cases, where one global value is mapped to a set of local values for one local interface, a set of URI is generated.

The process of a URI building can be:

```
String uri = "http://www.caseumc.com/header.asp?";
uri += "sltModl=All&hdnPreviousProximityPage=";
uri += "&sltManu=" + manufacturermapping;
uri += "&txtMeterFrom=" +
        ((hoursfrom.equals("")) ? "0" : hoursfrom);
uri += "&sltToYear=" +
        ((yearto.equals("")) ? "2006" : yearto);
uri += "&sltFromYear=" +
        ((yearfrom.equals("")) ? "1950" : yearfrom);
uri += "&HP=Any";
uri += "&sltProd=" + typemapping;
uri += "&hdnLingo=1&txtMeterTo=" +
        ((hoursto.equals("")) ? "99999" : hoursto);
```

Figure 4.11: A part of a URI builder. An example of constructing a URI. The variables are already mapped and transformed parameter values. The parameter names are fixed.

- manual, where a special Java class is created that implements an interface. The interface contains one method that returns a list of URIs.

  For an example see Figure 4.11.

  This is a problematic part that brings high requirements into a process of creating a scenario. A developer has to know a programming language and the deployment process is relatively complicated. Any change requires a restart of the whole MetaSearch server.

- automatic, actually a quasi-automatic process. A developer can use a GUI in the administrator console to create visually a mapping between global and local parameters.

  Unfortunately in practice this method was not usable for complicated portals. A session identification meant that a manual URI building process was necessary with a special pre-loading of a local source page where a session id was obtained.

4.5.4 Technology

The implementation of the current version of MetaSearch started almost three years ago and the older design and technologies used there can not meet increasing requirements.

Also new technologies matured and allowed the implementation of otherwise hard to implement features in the old version.

A good example is the scalability feature, that can be easily achieved by using an enterprise technology such as J2EE.

The current implementation uses J2SE and hence the environment does not provide any support.

The main problems of the current version are listed in the next sections. The last section summarizes new requirements.

4.5.5 Intermediate Results

MetaSearch retrieves results from more sources, each of them has its characteristic such as how fast the server is or the complexity of the wrapper. The implication of characteristics is a different response time of each source.

The current version, as shown in Figure 4.6, starts all retrieving processes in a parallel mode, however it waits until all of them are finished and then performs a transformation of all particular results to one unified result and sends it.
Obviously all characteristics of all sources are disregarded.

For a non-human recipient this drawback may not be desirable, but for a human recipient, when the showing of results can be done progressively, by using technologies such as AJAX, this drawback is critical .
For an example see the A.1 and A.2 figures on pages 144 and 145.
It is an interesting feature that improves usability and responsiveness.

Also non-human recipients can take advantage of this feature; a recipient can stop the provision of next results after several received results, e.g. it needs only a limited amount of data or data to comply to a condition.

4.5.6 Multiple Requests

MetaSearch was designed in a way that enables it to handle multiple requests.
It receives requests so often that many processes of the previously received request or requests are still running. This, of course, increases processor load and memory consumption.
Slow responsiveness is the main consequence, which for a commercial meta search application is critical and unacceptable.

The tests mentioned below were performed on a Dell desktop computer with Intel Pentium 4 3 GHz, 2 GB RAM and the Linux operating system. MetaSearch was tested with the Lixto FlightSearch project that searches on several flights pages for the best prices.
The *JMeter* is a testing application, for more details see the [Apa06a] page, that is used for sending requests and measuring times.

Concurrent Requests Test

The *concurrent requests* test results are shown in the 4.1 table.
All requests were sent at the same time. The *total response* time means the response time for all sent requests.

The tests are commented in the following list:

1. MetaSearch was fast enough for the *one request* test.

   The response time is 11 seconds that is a user-friendly value.

2. MetaSearch was acceptably fast enough for the *six requests* test.

   The start time is 42 seconds, but it can be hidden for an end user with an artificial initial request to the server.

Config	CPU [%]	Mem [MB]	Requests	Startup [ms]	Response [ms]
6-3-3	50%	256	1	7	11
6-6-6	100%	552	6	42	35
10-10-10	100%	952	10	300	91

Table 4.1: The results of the multiple requests test. Legend: *Config*: Configuration (the maximal number of concurrent requests – the size of the downloading processes pool – the size of the extracting processes pool), *CPU*: Percentage CPU usage, *Mem*: Memory usage in MB, *Requests*: Number of Requests, *Startup*: Startup time in ms, and *Response*: Total response time in ms.

The total response time is acceptable, although a waiting page or message should be displayed for the end user, for more details about usability see the [Nie94] book.

3. The *ten requests* test shows that MetaSearch is unusable for ten concurrent requests on relatively strong hardware.

   An end user would not wait $1\frac{1}{2}$ minute for a result, as defined in the [Nie94] book.

   The server was totally unserviceable[1] with this configuration under the load.

Success Test

The *success test* is another test that checks the ability of a server to respond to a set of sequential requests sent in a specified interval.

The results are shown in the 4.2 table and the 4.12 figure.

The most suitable *6-6-6* configuration from the previous test was used in this test.
It emerges from the results that relatively strong hardware allows the sending of a request every seven seconds.

The 100% success rate is guaranteed only if the time interval between requests is 7 seconds or more. Requests sent in shorter time intervals will be refused.
An improvement of the responsiveness is an interesting feature for an installation that will be accessed by a higher number of users.

The results of the concurrent requests test and the success test imply the need for scalability and the new design will focus on processing higher loads.

4.5.7 Session Identification Assignment

The session identification assignment problem was already mentioned in Section 4.5.3.

[1] The system load average was higher then 11.

Time [sec]	Successful Requests	Success Rate
1	3	30%
10	3	30%
20	4	40%
30	6	60%
40	8	80%
50	8	80%
60	9	90%
70	10	100%

Table 4.2: The results of the success test.

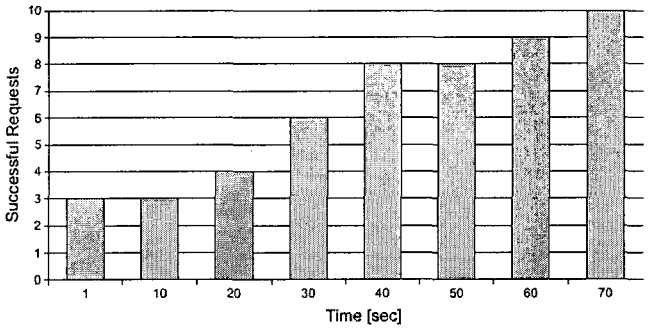


Figure 4.12: The results of the success test.

Complicated web pages or portals, mainly generated by a web application frameworks, such as Struts, Sun Portal, and Apache Tomcat, use a session identification, for more details see their pages [Apa05a], [Sun05d] and [Apa05b] about usage a session id.

A session identification is usually a special parameter in an HTML GET or POST request, or it is stored in a site cookie.

Because the current MetaSearch version generates all URIs for a local source in one step in a URI builder, there is only one way to obtain a session ID – to make a special request to the local source in the URI builder, where a session ID is extracted and then inserted into all URI generated by this URI builder.

This approach caused another problem in the current MetaSearch version.
When a request is received, MetaSearch generates URIs for all local sources. This part is sequential. The generated URIs are processed in a parallel, i.e. multi-threaded, part.
Special requests meant that the sequential process of creating URIs was significantly slower.

A fix for the current MetaSearch version split a URI builder into two parts, the first one is called in the sequential mode, the second in the parallel mode. A developer can decide which part to use.

The fix is an artificial element and broke the designed architecture of the current version.

4.5.8 Wizard Search Interfaces

The problem of wizard search interfaces is similar to the session identification assignment problem, but it is basically not solvable in the current MetaSearch version.

A wizard search interface was explained in Section 3.1.

If a transition between pages is provided through only a session identification, the solution described in the previous section will help.

If a transition between pages uses generated parameters, e.g. for saving values entered in previous pages, there is practically no solution.

Generated parameters can be generated by a know algorithm such as a simple copy of entered values. A more complicated algorithm can be *hacked*, but it this method can not be called a general solution.

Also an artificial parameter like a time stamp can ruin an attempt to *emulate* a step.

A change of the algorithm on the local server side will not only make a developed scenario unusable, but this change is also hard to detect compared to the detection of a changed parameter.

4.5.9 New Requirements

Almost all the following requirements of the new version are missing and hard to implement features of the old version:

- Faster development of MetaSearch elements. There should be an easier way to develop elements needed for a scenario like mappings. The currently used creation of Java classes requires compiling, that is not trivial for the majority of users.

  The simplification consists of two parts: a graphical user interface, that is partially designed in Section 6.3.6, but an implementation is not covered, and the using of scripts for mappings, that is designed in Section 5.5.4 and Section 5.10 and an implementation is described in Section 7.3.3.

- Monitoring of the whole process. A checking of the health of system should be available.

  A design is described in Section 6.3.5 and Section 6.4.5, an implementation is shown in Section 7.5.

- Scalability and load balancing. The solution should be scalable to meet the requirements of a customer.

  A design is described in Section 6.5.3, an implementation in Section 7.3.1.

- Caching. The system should cache results to speedup the responsiveness of the system.

  A design is shown in Section 6.1.4 and an implementation is Section 7.3.1.

- Source Pruning. A source pruning should be a part of the searching process. Sources that did not provide any result for a similar query will be searched at the end of a searching process, or not at all.

  A partial design is shown in Section 6.1.4. The implementation is not covered.

- Intelligent Pre-loading. The most requested sources and queries should be pre-loaded to speedup the responsiveness of the system.

  A partial design is shown in Section 6.1.4. The implementation is not covered.

- Accessible Intermediate Results. Intermediate results of a search process should be available, not only at the end of the whole search process.

  A design is described in Section 6.3.4 and Section 6.5.2, an implementation is shown in Section 7.3.1.

- Session ID Assignment. A session identification assignment should be performed automatically, without any artificial steps. A developer should not perform any special steps to handle a session identification.

  A solution is explained in Section 5.9.

- Multi-page search forms. Multi-page or the wizard style of search forms should be supported without any special handling. A developer should not perform any additional steps for handling search forms; generated internal parameters should be handled automatically.

  A solution is explained in Section 5.9.

In general, all requirements are designed in Chapter 5 and Chapter 6.

Chapter 5

Form Mapping Framework Design

A mapping between global and local elements is the fundamental operation in MetaSearch. This chapter explains how the form mapping theory is used in MetaSearch.

A notation of a mapping will be defined, where a good choice of the notation will bring the desired flexibility to the whole process.

The first section explains the form descriptor. The next two Sections *Form Mapping* and *Reverse Mapping* describe the design of the form mapping theory in MetaSearch. *Mapping Design* Section shows different notations of a mapping. The two sections *URL Mapping* and *Navigation Mapping* contain two technical realisations of the mapping design.

The theoretical background is explained in Chapter 3.

The main requirements naturally emerge from the theory – to map global and local parameters and values in different mapping styles; the experiences from the solution mentioned in Section 4.5, Lixto MetaSearch, determine the requirement for easier and faster development of mappings mentioned in Section 4.5.9.

5.1 Form Descriptor

The form descriptor is an image of a search form. It is an excerpt of a search form that contains:

- parameter names
- parameter values

The structure of the form descriptor has the *descriptor* root element with a unique identification, *name*, of the corresponding local search form.

The root element contains the *param* element that may contain the *value* elements with values, if the parameter is a list of options, the values are extracted and the *list* attribute with the *on* value is added.

The structure is shown in Figure 5.1.

```
<descriptor name="localForm1">
        <param name="paramName1"/>
        <param name="paramName2" list="on">
                <value>paramName2Value1</value>
                <value>paramName2Value2</value>
        </param>
</descriptor>
```

Figure 5.1: The structure of the form descriptor.

All techniques and algorithms described in the next sections work with a set of form descriptors for all required local search engine forms.

Figure 5.2 contains an example of a form descriptor. Original search forms are shown in Figure 8.5.

5.2 Form Mapping

The form mapping theory mentioned in Chapter 3 is applied in the development phase.

Advanced techniques such as semantic matching or weight-based matching support a developer in the following steps:

- finding an appropriate mapping between global and local attributes and their values.
- defining a global interface.

Form descriptors of at least two local search engines are required for the above-mentioned operations.

```
<td class="sub" valign="top" width="28%">
<select name="ADULT" size=1 class="sbsm">
<option value="0">0
<option value="1" selected="selected">1
```

```
<param name="adult" list="on">
<value>0</value>
<value>1</value>
```

Figure 5.2: The first part shows the adult parameter in the Ryanair [Rya06] search form HTML page. The second part shows the corresponding form descriptor part.

5.3 Reverse Mapping

The reverse mapping is a mapping process, where global attributes and their values are transformed to local parameters and values that a local search engine understands.

This process is performed for every local search engine. A form descriptor is required for this process.

In the reverse form mapping process the same approaches can be applied as in the form mapping, just in the reverse meaning.

MetaSearch uses the reverse mapping during processing requests, that is a mapping from global to local attributes and values is used in the processing of requests.

5.4 Mapping Notation

A mapping notation describes a matching relationship between global and local attributes – their names and values. For each local source there is a mapping notation between the global attributes and the local attributes defined in the local source.

In the next sections various relationships are defined, from simple such as *complete notation* to the most complicated *script notation.*

All notations are described in the XML style. This does not have any impact for the final implementation, where another technologies can be used.

Because all notations are XML based, all of them have to have some common elements that are also shown in Figure 5.3:

- Root Element. The *mapping* root element is shown in line 1. It contains the *id* attribute that contain a unique identification for a mapping.
- Global Parameter Element. The *global* element is shown in lines 2 and 11. It contains the *name* attribute that points to a global parameter name.

  A global parameter element contains two kinds of children elements: a specific mapping notation for the parameter mapping, in line 3, or the *value* element that is described below.
- Global Value Element. The *value* element is shown in lines 4 and 7. It contains the *name* attribute that points to a global parameter value name.

  Its children elements are particular mapping notations; it is shown in lines 5 and 8.

Every mapping notation has the following form that is also shown in Figure 5.4:

- A mapping element and its attributes. This is specific for every notation.
- A set of the *local* elements, where each of them contains only an attribute *id* that points to a local search engine, that is the mapping notation will be used to transform global parameter or value to a local one for the specified local search engine.

```
<mapping id="mappingId">
        <global name="globalParam1">
                <!-- parameter mapping -->
                <value name="globalParam1Value1">
                        <!-- value mapping -->
                </value>
                <value name="globalParam1Value2">
                        <!-- value mapping -->
                </value>
        </global>
        <global name="globalParam2">
                <!-- detto -->
        </global>
</mapping>
```

Figure 5.3: A structure of common elements for mapping notations.

```
<notation-name name="localParam1">
        <local id="localId1"/>
        <local id="localId2"/>
</notation-name>
```

Figure 5.4: A structure of local elements for mapping notation.

The normalisations mentioned in Section 3.3.2 are available for notations. By default the following of them are used: lower case and only alphanumeric characters.

Whether or not normalisation can be used is specified by attributes in the notation tag: *lower-case*, *only-alphanum* and *trim*. For a schema see Figure 5.5.

```
<notation-name name="localParam1" lower-case="true"
        only-alphanum="true" trim="false">
        <!-- local elements -->
</notation-name>
```

Figure 5.5: A structure of a mapping notation with attributes for normalisation.

Obviously all local parameter names and their values have to correspond to their local form descriptors.

5.4.1 Complete Notation

The complete notation is the most straightforward notation. It simply specifies the exact mapping between two terms, i.e. a global term is replaced by the specified local term.

The complete notation has the following structure:

- Root element *complete*.

```
<global name="adult">
        <complete name="adult">
                <local id="ryanair"/>
        </complete>
</global>
```

Figure 5.6: The complete mapping notation for the adult parameter of the Ryanair [Rya06] search form.

```
<lists>
        <synonyms id="synonyms1">
                <item>synonym1</item>
                <item>synonym2</item>
        </synonyms>
        <!-- etc -->
</lists>
```

Figure 5.7: The semantic list with synonyms structure.

  - Attribute *name* that defines the result term.

- Children elements *local* that were explained before.

For example see Figure 5.6, where the global parameter *adult* is mapped to the local parameter *adult* for the *ryanair* local source.
Original search forms are shown in Figure 8.5.

5.4.2 Semantic Notation

The semantic notation, described in Section 3.3.1, is a flexible mapping, where a change of the name of a parameter or a value can still be handled without a revisory intervention, if the new value is already defined in a semantic notation.

The semantic notation has the following structure:

- Root element *semantic*.

  - Attribute *synonyms*, *hypernyms* or *meronyms* that defines which list of semantic relationships will be used.

- Children elements *local* that were explained before.

A list of semantic relationships is defined globally, so it can be reused in more cases.
Its schema is shown in Figure 5.7. A root of lists is the *lists* element, that is a child of the root element. Every kind of semantic relationship has its list root element – *synonyms*, *hypernyms* or *meronyms* with the *id* attribute; its value is used to identify the right semantic list in a semantic notation.

```
<value name="Austria">
        <semantic synonym="country-austria-synonyms">
        <local id="Source1"/>
        <local id="Source2"/>
        </semantic>
</value>

<lists>
        <synonym id="country-austria-synonyms">
                <item>Austria</item>
                <item>A</item>
        </synonym>
</lists>
```

Figure 5.8: The semantic mapping notation example.

The semantic mapping allows a user to specify a list of related words in terms of semantics, e.g. synonyms, for a parameter name or value. The mapping framework will automatically find the appropriate value for a particular source described by its form descriptor that contains the value.

For example see Figure 5.8, where the global value *Austria* is mapped by the semantic mapping *country-austria-synonyms* to two possible values *Austria* and *A*.

The mapping framework then checks the local form descriptors if they contain one of the synonym values, e.g. the *Source1* source contains the value *A* and the *Source2* source contains the value *Austria*, and automatically maps a global value to right local values, i.e. the *Austria* value for the *Source1* source and the value *A* for the *Source2* source.

5.4.3 Weight-based Rule Notation

The weight-based rule notation allows to use mapping techniques described in Section 3.3.2.

The weight-based rule notation has the following children elements and attributes that are shown in Figure 5.9:

- Attribute *metric* that identifies the used metric and attribute *threshold* that specifies the threshold of the metrics. The attribute *name* was explained before.

  The following metric identifiers are defined:

  - ed – edit distance
  - vss – vector space similarity
  - bp – boolean property. The use of this metric requires two additional attributes *true-value* and *false-value*, like `<weight name="param1" metric="bp" true-value="on" false-value="off">`.

- Children elements *local* that were explained before.

```
<weight name="localParam1" metric="metric_type" threshold="t">
        <local id="localId1"/>
        <local id="localId2"/>
</weight>
```

Figure 5.9: The weight-based rule notation structure.

5.4.4 Combined Notation

The combined notation represents a comfort notation that combines all above-mentioned notations to allow a user to use an appropriate notation for parameters, their values and local search engines independently.

As shown in Figure 5.10, a parameter or value can be mapped for different local search engines independently.

```
<complete name="localParam1">
        <local id="localId1"/>
        <local id="localId2"/>
</complete>
<semantic synonym="syn1">
        <local id="localId3"/>
</semantic>
```

Figure 5.10: The combined notation structure.

If there are more notations for the same local search engine identifier, only the first one is considered.

5.4.5 Value Mapping Notation

The value mapping notation is the same as for the parameter names notation. Special cases like mapping of a list are mentioned below in Section 5.5.

Parameter mapping without value mapping means a value of the parameter is copied without any change.

The full and complete mapping structure is shown in Figure 5.11.

5.5 Advanced Mapping

This section shows how to map special cases such as a list of values and a date.

5.5.1 List Mapping

The list mapping is a mapping of a list of values, therefore it can only be used for values.

```
<mapping id="mapping1">
        <global name="country">
                <complete name="country">
                        <local id="Local10"/>
                        <local id="Local20"/>
                </complete>
                <value name="United_Kingdom">
                        <complete name="UK">
                                <local id="Source1"/>
                        </complete>
                        <complete name="United_Kingdom">
                                <local id="Source2"/>
                        </complete>
                </value>
                <value name="Austria">
                        <semantic synonym="country-austria-synonyms">
                                <local id="Source1"/>
                                <local id="Source2"/>
                        </semantic>
                </value>
        </global>
        <lists>
                <synonym id="country-austria-synonyms">
                        <item>Austria</item>
                        <item>A</item>
                </synonym>
        </lists>
</mapping>
```

Figure 5.11: An example of the full structure of a mapping notation. It shows all described notations and elements in one structure.

The mapping does not require a special notation – it is performed automatically if one side of a mapping is a list or both sides of a mapping are lists, which is defined in the corresponding form descriptors.

The mapping has two features:

- It checks if a value for a list was found, otherwise the mapping is not successful.
- It provides an index of the mapped value for the target form with a list. If the target form does not contain a list, just the mapped value is provided, i.e. the mapping has the same behaviour as other mapping.

There are three possible situations in the list mapping:

- A global interface contains a selection list and a local interface does not.

  In this case a global interface is more restrictive then local interface. A mapping has to contain pairs: a global interface selection list item – a local interface value. A local interface may be capable to return results also for values other than in a global interface selection list.

- Both interfaces contain a selection list.

  A mapping contains pairs: a global interface selection list item – a local interface selection list item. A mapping may not link all items from a global to a local interface and vice versa.

- A local interface contains a selection list and a global interface does not.

  A mapping contains pairs: a global interface value – a local interface selection list item. This situation indicates a wrong design, because it allows a user to enter a value that the corresponding local interface can not handle, i.e. it is not in the list.

5.5.2 Date Mapping Notation

The date mapping notation is based on the date value merging defined in Section 3.4.1.

The date format is the same as in the above-mentioned section.

For the structure see Figure 5.12.

5.5.3 Currency Mapping Notation

The currency mapping notation is based on the currency value merging defined in Section 3.4.1.

The format is the same as in the above-mentioned section.

For a structure see Figure 5.13.

5.5.4 Script Mapping Notation

The script mapping is a pragmatic approach to the complexity of the real HTML world. It allows to evaluate a script that computes the value of a local parameter from the value of the corresponding global parameter or it computes the whole mapping of all or a part of parameters and their values.

```
<date-split format="date_format" get="date_part">
        <local id="localId1"/>
        <local id="localId2"/>
</date-split>
```

```
<date-concatenation format="date_format">
        <day>
                <!-- parameter mapping -->
        </day>
        <month>
                <!-- parameter mapping -->
        </month>
        <year>
                <!-- parameter mapping -->
        </year>
        <local id="localId1"/>
        <local id="localId2"/>
</date-concatenation>
```

Figure 5.12: The date mapping notation structure.

```
<currency-split format="currency_format" get="currency_part">
        <local id="localId1"/>
        <local id="localId2"/>
</date-split>
```

```
<currency-concatenation format="currency_format">
        <currency>
                <!-- parameter mapping -->
        </currency>
        <number>
                <!-- parameter mapping -->
        </number>
        <local id="localId1"/>
        <local id="localId2"/>
</date-concatenation>
```

Figure 5.13: The currency mapping notation structure.

A script represents a logic necessary to transform a set of values to a set of values. The background can be technical, such as adding leading zeros, or business, such as making a currency conversion.

All the used scripts are defined, like semantic relationship lists, as children of the *scripts* element, that is a child of the root *mapping* element. For the structure see Figure 5.14.

```
<scripts>
        <script name="script name">
                script commands
        </script>
</scripts>
```

Figure 5.14: The structure of a list of scripts.

All examples are written in the JavaScript language. For more details see Section 5.10.

This mapping can be called in two ways: value or parameter script, i.e. to compute a value of a parameter or to compute the name of a parameter optionally with its value.

Value Script

A value script is a simpler usage of the scripting functionality. It is used as one of the value mapping notations and allows to use a script to compute a value.

It can compute a specific global value that is specified in the *name* attribute of the corresponding *value* node, which is used mainly for textual values; or it can compute any value where the *name* attribute is kept empty, which is used mainly for numeric values, such as converting metric.

A value script has an access to all global parameters and values. It can only return a computed value.

The value script structure is similar to other value mapping notation structures, the *script* tag has the *name* attribute that contains a script name. The content of a script itself is stored in the above-mentioned list of scripts.

The example in Figure 5.15 shows a business logic that sums two values from the global search form to one value, because the local search form contains just one parameter – the sum of number of adults and children in the global form is the number of adults in the local search form. The *sumAdultChildren* value script sums the two values and is called for any value, because it is applicable to any number.

Original search forms are shown in Figure 8.5.

```
<global name="adult">
        <complete name="_ctl1:cboAdults">
                <local id="skyeurope">
        </complete>
        <value name="">
                <script name="sumAdultChildren">
                        <local id="skyeurope"/>
                </script>
        </value>
</global>
```

```
<scripts>
    <script name="sumAdultChildren">
                getGlobalValue('adult') +
                        getGlobalValue('children')
        </script>
</scripts>
```

Figure 5.15: The local script mapping notation for the adult parameter of the SkyEurope [Sky06a] search form.

Parameter Script

The parameter script notation is more complex than the value script notation. The structure is similar to the value script notation structure, it is only used for a parameter, i.e. as a child of the *global* node.

It can be called in the following modes:

- To Compute Parameter Name. A parameter script returns one text value to set the name of a parameter. This usage is similar to the value script notation.

  For example see Figure 5.16.

- To Compute Parameter Value Pair. A parameter script returns parameter name–value pairs. For example see Figure 5.17.

  This usage allows to compute sets of pairs and its strength is comparable to a URI builder in the current MetaSearch version. For more details see Section 4.5.3.

The script environment and support is mentioned in Section 5.10.

5.6 URL Mapping

Global attributes and their values are transformed to local attributes and values, that are mapped to local search engines URLs.

For example see Figure 5.18.

The URL mapping is suitable for simple search pages without any session identification or dynamic changing of parameters on a page, which are problematic for this mapping. On the other hand it is faster then the navigation mapping.

```
<global name="adult">
        <script name="getSkyEuropeParamNameAdult"
                <local id="skyeurope">
        </script>
        <value name="">
                <script name="sumAdultChildren">
                        <local id="skyeurope"/>
                </script>
        </value>
</global>
```

```
<scripts>
    <script name="getSkyEuropeParamNameAdult">
                '_ctl1:cbo' + 'Adults'
        </script>
</scripts>
```

Figure 5.16: The *local script mapping notation* for the adult parameter of the SkyEurope [Sky06a] search form.

```
<global name="adult">
        <script name="setSkyEuropeParamAdult"
                <local id="skyeurope">
        </script>
</global>
```

```
<scripts>
    <script name="getSkyEuropeParamNameAdult">
                storage.addParamValue('_ctl1:cboAdults',
                        getGlobalValue('adult') +
                        getGlobalValue('children'))
        </script>
</scripts>
```

Figure 5.17: The parameter script mapping notation for the adult parameter and its value of the SkyEurope [Sky06a] search form.

The mapping is available as the URL action in the Navigator tool, for more details see Section 4.3.1.

5.7 Navigation Mapping

Global attributes and their values are transformed to local attributes and values, that are mapped to parameters of search engine navigations.

This mapping is more powerful but slower then the previous one – a navigation mimics user actions, for more details see Section 4.3.1 and Section 6.5.3. It should be used for more problematic pages with the session identification, a JavaScript mapping of parameters or dynamic loading of content, e.g. pages build on the AJAX technology.

For example see Figure 5.19.

5.8 Comparison

There is a number of differences and new features between the new mapping framework and the existing mapping framework described in Section 4.5.3:

- There is only the value mapping in the existing mapping framework. The new one adds also mapping for parameters.
- The existing mapping framework has only direct mapping between values, the new one provides various mappings such as complete, semantic, list, date, and currency mapping.
- The existing mapping framework provides a possibility to create a URIBuilder, for more details see Section 4.5.3, that allows to use the Java programming language for complex mappings. The new framework provides the script mapping to achieve this functionality.
- The proposed mapping framework allows to combine script mapping with other mapping, i.e. to use an appropriate mapping where it is suitable. The existing framework does not allows to combine the direct mapping with the URIBuilder mapping.
- The proposed mapping framework simplifies the development and deployment process, because the scripting mapping does not require compiling – the URIBuilder mapping does.

The URIBuilder mapping and the script mapping notation represent a pragmatic approach to the complexity of the real HTML world. The newly added mapping notations, such as data and currency, reduce the need to use a complex script mapping notation for common mapping of date and currency parameters.

The URIBuilder mapping is a URL mapping, i.e. a URIBuilder produces a URL. The proposed mapping framework allows to use two mappings: the already existing URL mapping and the new navigation mapping.

The navigation mapping in the proposed mapping framework solves many difficult problems that occurred with the URL mapping, e.g. JavaScript mapping of parameters or session identification.

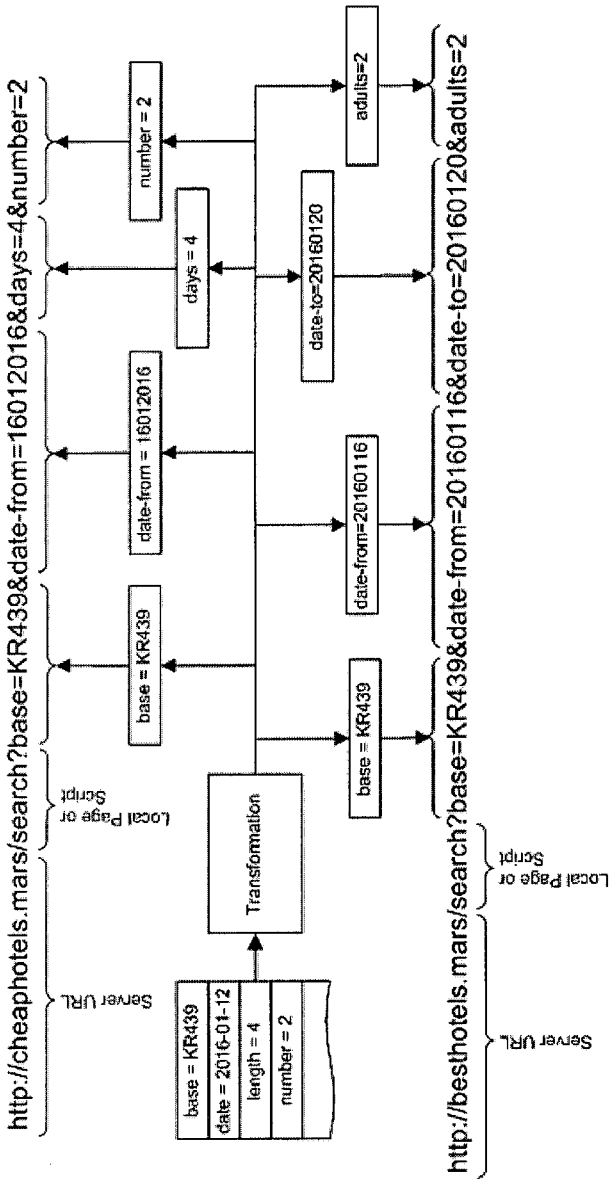


Figure 5.18: An example of the URL mapping. Input parameters and their values are transformed to the suitable form for two search engines.

Figure 5.19: An example of the Navigation mapping. Input values are mapped to parameters of navigations.

5.9 Mapping Strategies

The URL mapping was the only possibility of mapping in the current MetaSearch version. It required special handling in cases like multiple page forms or session identification mentioned in Section 4.5.7 and Section 4.5.8.

The main idea and basically also a solution to the above-mentioned special cases is to use the new navigation mapping that handles these cases naturally, because a standard browser is used. See Section 4.3.4 and Section 6.5.3 for more details.

5.10 Execution Language

An execution language support adds the possibility to write custom mapping notations and perform complex custom operations.

An execution language is a standard programming language with features such as variables, conditions, cycles, functions, and recursion. This allows the writing of mapping logic limited only by a programming language.

There are many scripting languages for Java, summarised on the [VM02], [Jav02] and [Jav06b] pages, with different features that can be used as an execution language.
The *Rhino* scripting library, described on the [Moz06] project page, that supports the *JavaScript* language was chosen for its similarity to Java and its easy to use.

Because the scripting environment is a complex feature, it is facilitated by a scripting framework.
Unfortunately scripting frameworks technology background is not compatible with the technological background of the execution part of the solution. A similar problem is described in Section 6.5.3 and technical details are mentioned in Section 4.4.3 and Section 7.3.
The solution is based on the same principles as the *Hydra* solution mentioned in Section 4.4.3. They together represent a complete solution: the new component called *Octopus* allows the running scripts as *Hydra* allows the starting of the downloading and extracting tasks.

The Hydra scripting framework supports the JavaScript scripting language but it is ready to be extended also to other scripting languages such as Jython or JRuby as well.

Chapter 6

Execution Framework Design

This chapter shows the design of the *Snorri* metasearch system. The standard object-oriented software development techniques are used.

The requirements are listed and explained in the first section. The actors are defined in the second section. The third section defines use cases for each actor defined in the first section.

The last sections design the Snorri metasearch system based on the previous sections.

6.1 Requirements

The Snorri metaseach system is a distributed data flow system capable of receiving and processing requests, storing and providing results, and optimising the whole process according to the specific and characteristic area.

It has to comply with various requirements; from simple and straightforward requests such as to accept a request to complex requests such as distributed processing.

6.1.1 Vocabulary

The first step in a design is to define a vocabulary to understand all terms.

- Parameter. A pair of a name and a value is a *parameter*. It is denoted by $P(name, value)$, e.g.

  $$P(\text{“artist”}, \text{“Leonardo da Vinci”})$$

- Request. A set of parameters is called a *request*. Basically the parameters in a request are parameters of the general interface. They will be mapped to particular interfaces. A request is denoted by $R(P_1, P_2, .., P_n)$, where P_i is a parameter, e.g.

  $R(\text{"artist"}, \text{"Leonardo da Vinci"}), (\text{"artwork"}, \text{"Lady with an Ermine"}))$

- Result. Technically, a *result* is any information related to the specified request. It is denoted by $S(R)$, e.g.

$$S(R(\text{"artist"}, \text{"Leonardo da Vinci"}),$$
$$(\text{"artwork"}, \text{"Lady with an Ermine"})) =$$
$$\{\text{"Museum of Fine Arts, Houston, Texas"}\}$$

- Process. A function that transforms a request to a result. It is denoted by $f(R) = \{S(R)\}$.
- Request Producer. An entity that produces requests.
- System. An entity that by using a process for a request produces results is a *system.*
- Source. A source of information. E.g. a museum web page.
- Scenario. A set of sources that provide information from the desirable sphere. E.g. museum web pages, airline pages or book shop pages.

6.1.2 Basic Requirements

The *basic* requirements are:

- Non-GUI:
  - Accepting Request. The system is accepting requests.
  - Confirming Accepted Request. The system sends a confirmation for an accepted request.
  - Providing Result. The system provides the obtained result from a request.
  - Providing Partial Results. The system provides also partial results for a request. I.e. if the processing of a request reads more sources, like more book shops, the results are available immediately for each source. *This is one of the main improvements of the new version.*
  - Harmonising Result Data. Result data are from various sources in different formats and styles. It is necessary to harmonise them, i.e. to transform them into a common format and style.
  - Sort Result Data. Result data are in most cases in a tabular form and they are sorted by a predefined column.
  - Session Identification Assignment. A session identification assignment should be done automatically.
  - Multi-page Search Forms. Multi-page or the wizard style of search forms should be supported without any special handling.
  - Responsiveness. A result should be sent in less then 7 seconds (at least partial result).
  - Meta-data. An output contains the result and result meta-data. Meta-data contain information about processing like times, possible warnings or error descriptions.

- GUI:
  - Sort Result Data. An end-user has an opportunity to change the sorting direction and the sorting column.
  - Scenario Developer Tool. A developer tool for creating scenarios. Should contain a mapper tool and a general form generator. These requirements were already specified in the [Ros04] thesis.

    A detailed design and an implementation are not covered in this thesis.

6.1.3 Advanced Requirements

The *advanced* requirements make the whole solution more comfortable for use, stable and powerful:

- Monitoring. The system maintains a list of running processes and gives a possibility to display and control them. *This is one of the improvements of the new version.*
- Distribution & Reliability. The system can be distributed on more servers. The distribution brings a better reliability. *This is one of the main improvements of the new version.*
- Easy Developing. An easy developing of mappings. *This requirement is not covered in this thesis.*
- Deep Linking. Retrieved results should contain a link that points to a link on the original page. It allows to an end customer make a booking or buy an item in the result.

6.1.4 Optimisation Requirements

The *optimisation* requirements speedup a response time and save a network traffic:

- Caching Results. The results of a request are saved to speedup a response time of the next requests with the same parameters as the parameters of the first request.

  The main idea is that users tend to have similar requests. E.g. in a book shop scenario people tend to look for bestsellers.
- Reasonable Sources. When a process for a request uses more sources, statistics of the successfulness are computed and stored. A source, that returned any information is more *reasonable*, and vise versa.

  A next similar request, i.e. with similar parameters, is processed more optimal, because the more reasonable sources are used by the process as the first, the less reasonable sources are processed as the last or not at all.

  The main idea is that sources tend to provide similar responses. E.g. for a flight search scenario airlines tend to have regular flights, that is if an airline does not provide a regular flight from an airport, it will not provide it in the next days.

- Intelligent pre-loading. A technique that merges the previous techniques. The most used sources and the most reasonable sources obtained in the same way as in the previous optimisation technique are automatically processed without a user request. The results are then cached.

  The main idea is to predict possible requests and processed them in advance. The effect of this technique is a feeling for a user that the system behaves intelligently.

In general the optimisation techniques are dependent on specific scenarios.

The caching results technique can noticeably help to a *bookshop* scenario, because prices of books do not change every hour; on the other hand, a *flight search* scenario can use the technique partially, the results can be cached no more than tens of minutes.

6.2 Actors

From the previous section 6.1 the following actors are defined:

- Request Producer.
- Result Consumer.
- System.
- Supervisor.
- Scenario Developer.

6.2.1 Request Producer

It is a human being or an application that sends a request to the system.

The number of *request producers* can be from one to several tens, they may send their requests on a large-scale or in several hours, their requests may be sent periodically or randomly.

They may send an incorrect request, that is with parameter names not matching the general interface parameters or/and with a wrong parameter value.

6.2.2 Result Consumer

It is a human being or an application that is in a close relationship with the request producer.

It asks the system for a result or results of the previous request. It can ask periodically or randomly.

In most cases the Request Producer and Result Consumer actors will be the same person or application; the distinction is based on different goals – to send data and to receive data. These goals may require different technologies, user management, response times, etc.

E.g. a request producer is an application that periodically sends a request. A result producer is a simple application that asks for results, saves them in a database and publishes them on a portal accessible to registered user only.

6.2.3 System

The core component. It performes the processing and provides results.

6.2.4 Supervisor

It is a human being that checks the status of the system and running processes.

6.2.5 Scenario Developer

It is a human being that develops particular scenarios, that is he/she defines sources, general interface and mapping.

6.3 Use Cases

Use cases are a standard form of describing processes of actors.

A use case consists of several detailed steps that explain actor's actions to achieve the goal of a use case.

Use cases used in this section are slightly simplified – they may contain also conditions directly in the description of a use case.

There are use cases for each actor in the following sections. The first section explains the vocabulary necessary for the next sections.

6.3.1 Vocabulary

Use cases require new terms, that are defined in the following list:

- Result Handler. A unique identification of a set of results. It provides a unified medium for the result interchange between the *result consumer* and the *system* actors.

  For example it can be an integer number.

- Result Handler List. A list of result handlers stored on the *system* actor side. The list contains all result handlers that can be used in the result interchange between the *result consumer* and the *system* actors.

  If there cannot be any new result for a result handler, it is removed from the result handler list.

- Processing Queue. A queue of items that are going to be processed.

  All items have a priority. The topmost item will be processed as the first one. The item at the end of a queue will be processed as the last one.

  Items in a queue can be reordered according to any change of the priority of items.

  The processing queue mechanism allows to process limited and manageable number of items.

6.3.2 Request Producer

The list and Figure 6.1 show and explain use cases of the request producer actor:

- Produce Request.
  1. Prepare a set of parameters.
  2. Create a request from the parameters.
  3. Send it to the system. This will cause calling of the *accept request* use cases of the *system* actor.
- Ask For Confirmation. A confirmation contains a *result handler* that is necessary for the response consumer to identify the results.
  1. The *system* actor calls the *confirm request* use case. The use case sends a *result handler*.
  2. Receive a *result handler*.
- Exchange Result Handler.
  1. The received *result handler* is exchanged between the *result producer* and *result consumer* actors.

6.3.3 Result Consumer

The list below and Figure 6.1 show and explain the use cases of the result consumer actor:

- Exchange Result Handler.
  1. The received *result handler* is exchanged between the *result producer* and *result consumer* actors.
- Ask For Result.
  1. A *result handler* is sent in a result request message to the *system* actor.
  2. It causes, that the *provide result* use case for the *system* actor is called.
  3. Receive a result or an empty result if there is no new data.
- Stop Producing Result.
  1. A *result handler* in a stop producing result is sent to the *system* actor.
  2. It causes, that the *stop providing result* use case for the *system* actor is called.

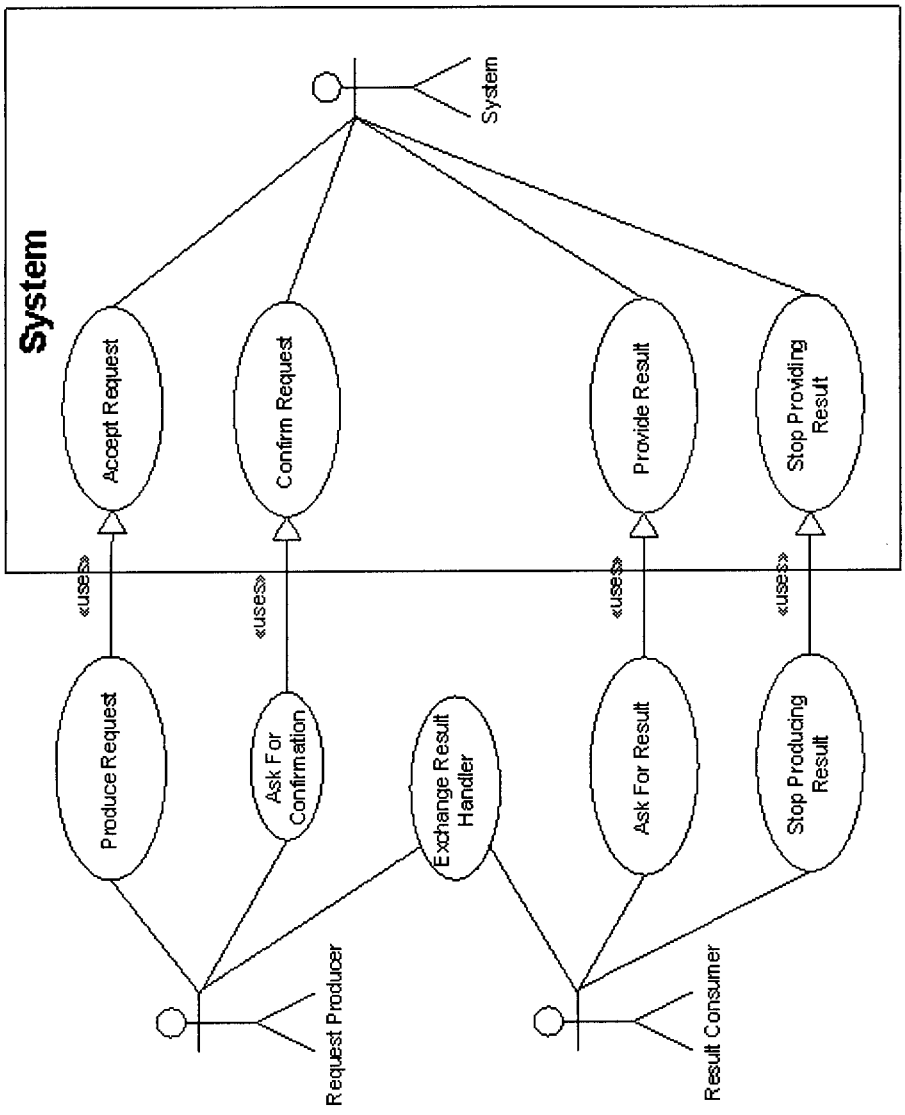


Figure 6.1: Use cases for the request producer and the result consumer actors.

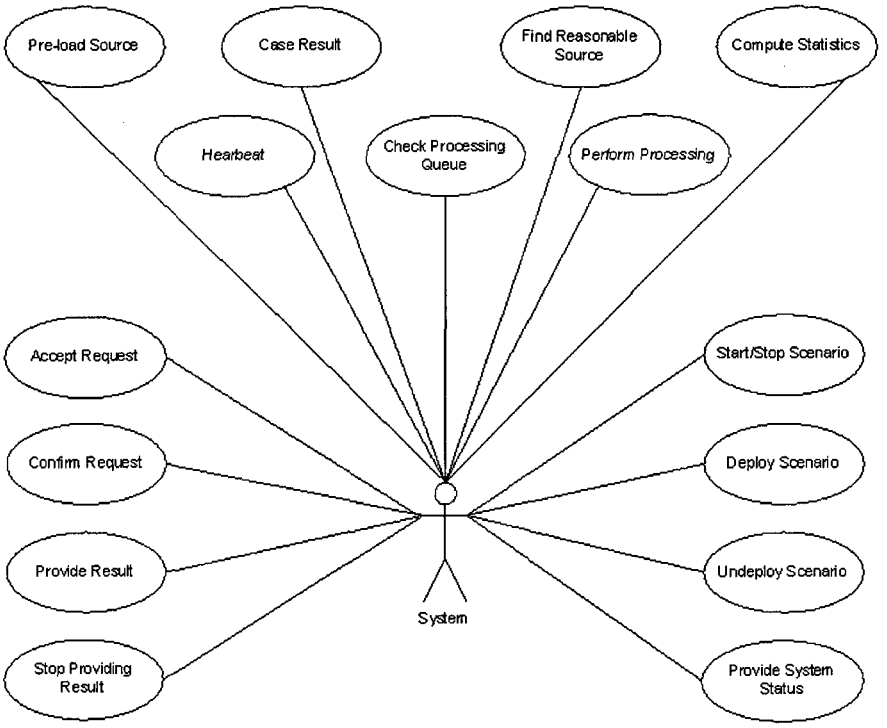


Figure 6.2: Use cases for the system actor.

6.3.4 System

The system actor is the most complex actor with the majority of use cases. Basically, it is the central actor of the solution.

Figure 6.1 and Figure 6.5 show user stories – interactions of this actor with other actors. They do not contain the complete list of all use cases of this actor.

The list below and Figure 6.2 show and describe all use cases of this actor; for simplicity the figure does not contain links to other actors:

- Accept Request. For a flowchart see Figure 6.3.
  1. Wait for an incoming request.
  2. Receive a request.
  3. Choose[1] the appropriate scenario.
  4. Check syntactic and semantic characteristics of parameters; parameters are correct.
  5. Create a *result handler*, that is an unique key that will identify all results that belong to this request.
  6. Register the created key to the *result handler list.*
  7. Take all parameters and store them.

[1] A technical solution based on a specific URL or via values of a special parameter.

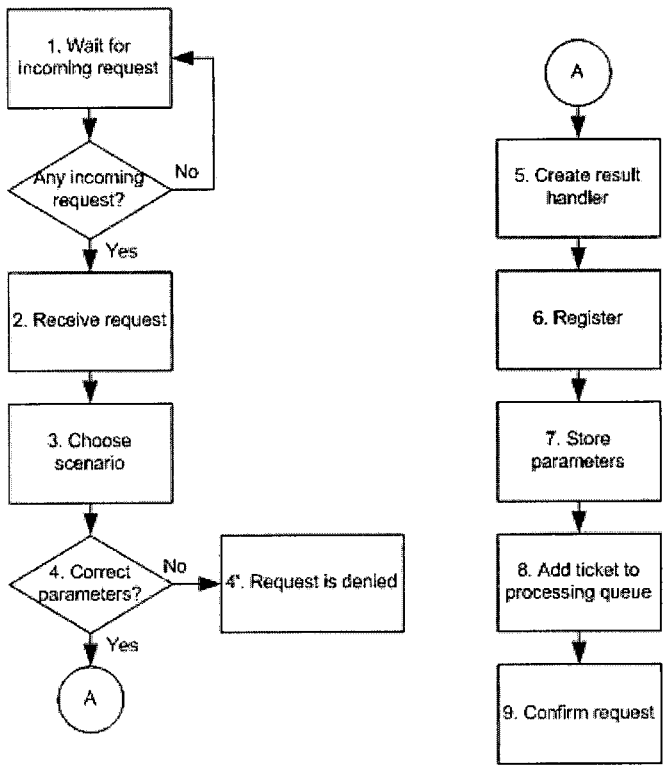


Figure 6.3: A flowchart for the accept request use case of the system actor.

8. Add a ticket to the *processing queue* for all sources.
9. Call the *confirm request* use case.

Alternatives:

4'. Check syntactic and semantic characteristics of parameters; if any of parameters is incorrect, the request is denied.

- Confirm Request.

1. Prepare a confirmation message, that is to create a message with the created *result handler*.
2. Send a confirmation message back to the *request producer* actor.

- Provide Result.

1. Check the *result handler list* for the specified *request handler*.
2. There is a stored result for the specified *request handler*.
3. Return it. Remove the returned result.

Alternatives:

2'. There is no stored result for the specified *request handler*.
3'. Return an empty response.

- Stop Providing Result.

1. The specified *request handler* is in the *result handler list*.
2. Stop all processes that are producing a result for the *request handler*.
3. Delete any stored results for the specified *request handler*.
4. Remove the *request handler* from the *result handler list*.

Alternatives:

1'. The specified *request handler* is not in the *result handler list*.
2'. – 4'. Return an error response.
3". There are no stored results for the specified *request handler*.

- Provide System Status.

1. Prepare and return the following list of system and business logic attributes:
   - uptime
   - system load
   - list of all/running/stopped/broken scenarios
   - list of running processes
   - list of delivered results
   - list of cached results
   - statistics

- Start Scenario.

1. Enable the specified scenario, that is it will be visible for the process of accepting a request.

- Stop Scenario.

  1. Disable the specified scenario, that is it will be invisible for the process of accepting a request.

- Deploy Scenario.

  1. Install the specified scenario to the system. This scenario is in the disabled state.

- Undeploy Scenario.

  1. Call the *stop scenario* use case for the specified scenario.
  2. Wait until all running processes of the scenario finish.
  3. Delete the scenario from the system.

- Cache Result.

  1. Store the specified request and result.
  2. Remove cached items older then a predefined threshold.

- Compute Statistics.

  1. Prepare the following list system and business statistics about:
     - number of received/processed/denied requests
     - number of processed/cached results
     - list of most used sources and parameters

- Find Reasonable Source.

  1. Check statistics for all sources.
  2. Take a predefined number of the most used sources and requests, that is sets of parameters, and store it in the *reasonable source* list.

- Pre-load Source.

  1. Take a group of most used sources with used parameters.
  2. Select a predefined number of them.
  3. Add a ticket to the processing queue for the selected sources with parameters.

- Check Processing Queue.

  1. The number of already running processes is smaller then a predefined threshold.
  2. Flag the ticket with the highest priority from the processing queue.
  3. Increase the priority for reasonable sources and vice versa.
  4. Increase the priority for cached results.

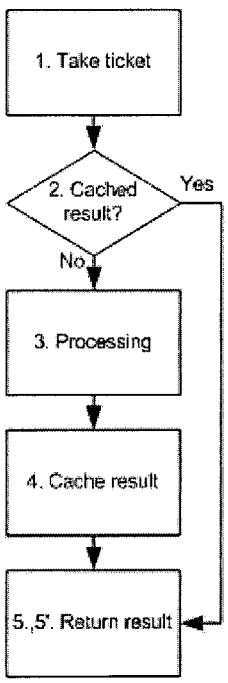


Figure 6.4: A flowchart for the perform processing use case of the system actor.

Alernatives:

1'. The number of already running processes is equal or higher then a predefined threshold.

2'. – 4'. Wait a predefined time.

- Perform Processing. For a flowchart see Figure 6.4.

  1. Take the flagged ticket from the processing queue.
  2. No cached result exists for the request.
  3. Send parameters from the request to the source, read a response, transform it if required, store it for the *request handler.*
  4. Cache the result (of the previous step), that is call the *cache result* use case.
  5. Store the result for the *request handler.*

  Alternatives:

2'. – 5'. Store the cached result for the request for the *request handler.*

- Heartbeat. This use case calls the specified use cases in a predefined intervals or timestamps.

  1. Call always the *check processing queue* use case.
  2. Call always the *perform processing* use case.

3. Call always the *compute statistics* use case.
4. Call sometimes the *find reasonable source* use case.
5. Call seldom the *pre-load source* use case.

6.3.5 Supervisor

The list below and Figure 6.5 show and describe the use cases of the supervisor actor:

- Monitor System Status.
  1. Retrieve the list of system and business statistics from the *system* actor. The statistics were prepared in the *compute statistics* use cases of the *system* actor.
  2. Display retrieved data.
- Control Scenario.
  1. Retrieve a list of all scenarios.
  2. Choose a scenario.
  3. Choose start, stop or undeploy operation.
  4. This will cause that use cases *start scenario*, *stop scenario* or *undeploy scenario* of the *system* actor are called.
- Exchange Developed Scenario.
  1. Transfer a scenario from the *scenario developer* actor to the *supervisor* actor.
- Deploy Developed Scenario.
  1. The scenario is transferred from the *exchange developed scenario* use case to the *system* actor.
  2. This will cause that the *deploy scenario* use case of the *system* actor is called.

6.3.6 Scenario Developer

The list below and Figure 6.5 show and describe the use cases of the scenario developer actor.

- Develop Scenario. For a flowchart see Figure 6.6.
  1. Design a scenario.
  2. Create a form descriptor for every local search engine.
  3. Create navigation for every local search engine. The navigation has to have parameters for necessary fields in the corresponding search form.
  4. Create a general interface descriptor. It contains a list of parameters and values accepted by a general search form.

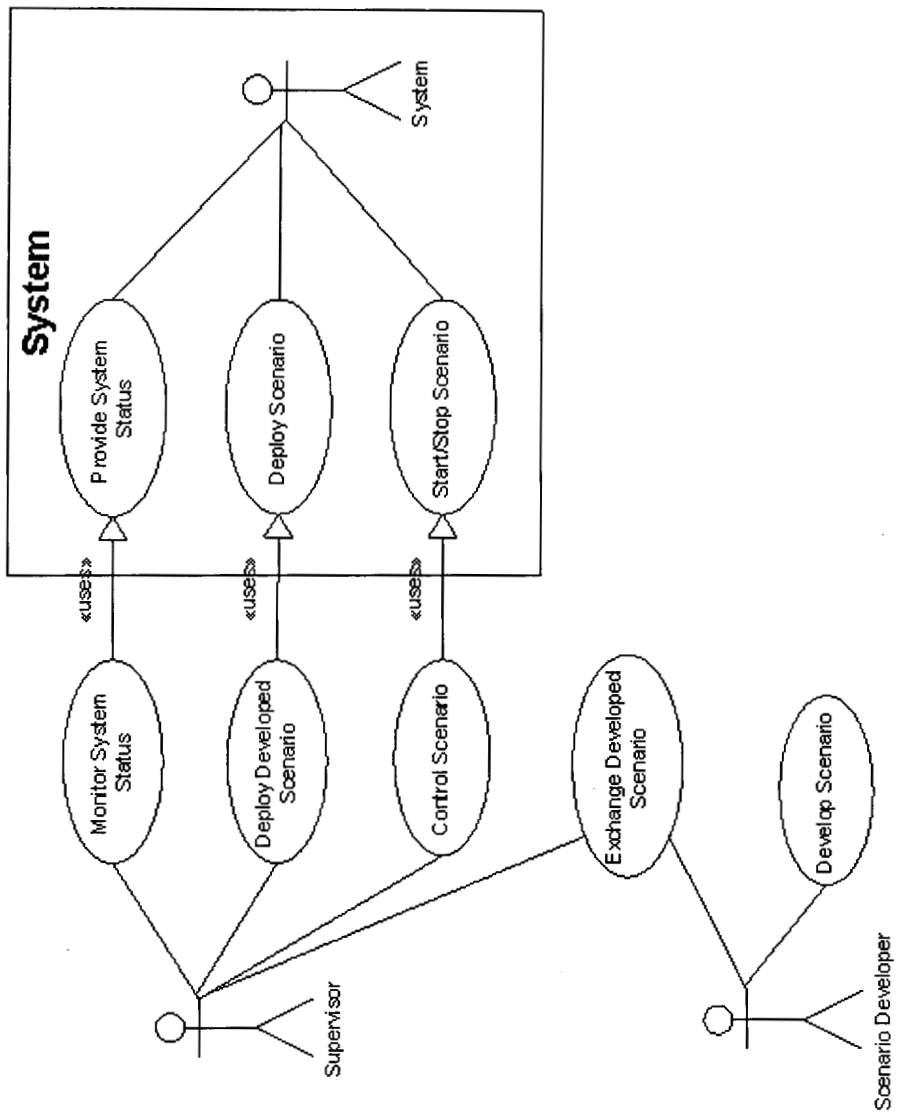


Figure 6.5: Use cases for the supervisor and the scenario developer actors.

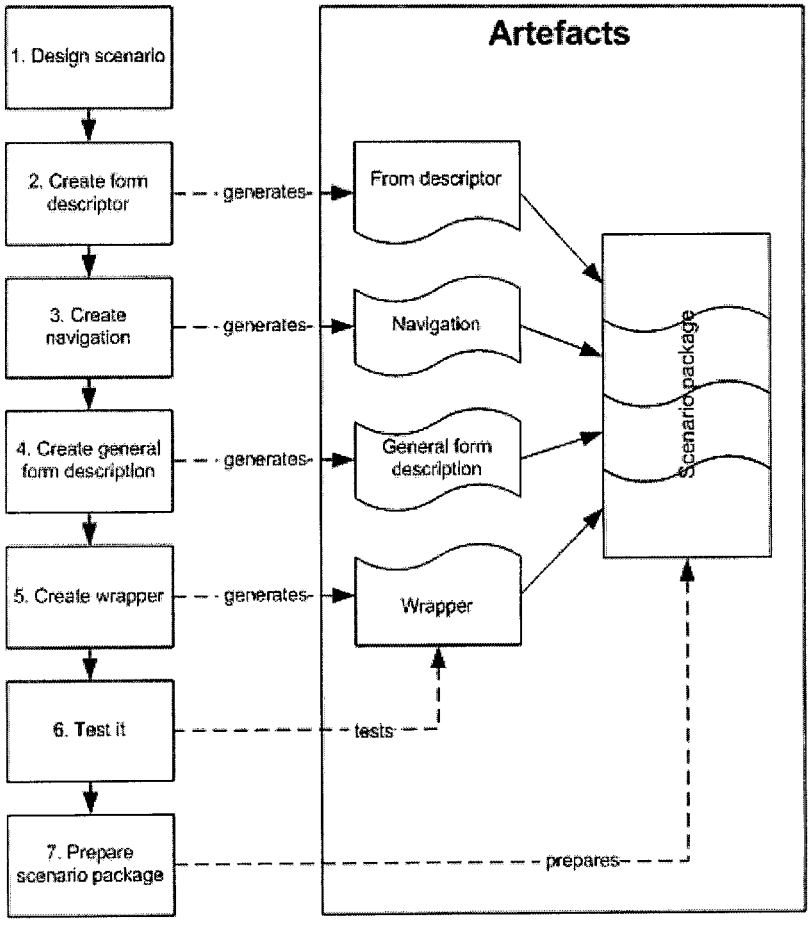


Figure 6.6: A flow chart for the develop scenario use case of the scenario developer actor.

5. Create a wrapper for results of every local search engine. A wrapper transforms results to an XML document.
6. Test it.
7. Prepare a scenario deploy package that will be used in the *exchange developed scenario* use case. A deploy package contains form descriptors, navigations and a general interface descriptor.

- Exchange Developed Scenario.
  1. Transfer a scenario from the *scenario developer* actor to the *supervisor* actor.

6.4 Components

Components represent logical units of functionality. Actors with similar functional fundamentals may belong to the same component. On the other hand, actors with a large group of use cases may be split into more components.

The Snorri metasearch solution consists of seven components:

- Engine.
- Manager.
- User Interface.
- User.
- Administrator Interface.
- Administrator.
- Developer.

The following sections describes the components

6.4.1 Engine

The most complicated, central, with the highest load – the core component. It merges all use cases of processing of requests, producing and providing results.

It complies with all basic requirements.

It has to comply with the *monitoring* and *distribution & reliability* advanced requirements.

And it has to comply with all optimisation requirements.

It is an environment for the following use cases of the following actors:

- System actor:
  - check processing queue
  - perform processing
  - heartbeat

- provide system status
- cache result
- compute statistics
- find reasonable source
- pre-load source

6.4.2 Manager

The manager component merges all use cases related to the managing of scenarios.

Although it provides descriptions as to *what to do*, i.e. scenarios, it is not a part of the engine component, because it is not a time nor processor critical part of the solution. Its needs are different and requires different approaches.

It is an environment for the following use cases of the following actors:

- System actor:
  - start scenario
  - stop scenario
  - deploy scenario
  - undeploy scenario

6.4.3 User Interface

The user interface component is an interface between the engine and the user component. It contains all necessary request and result handling use cases.

It complies to all basic requirements.

It is an environment for the following use cases of the following actors:

- System actor:
  - accept request
  - confirm request
  - provide result
  - stop providing result

6.4.4 User

The user component is one of the most variable components. It is based on scenario requirements and characteristics. In other words – it varies from scenario to scenario.

It uses features of the user interface component that in most cases can split the user component to two parts:

- Generating Request. This part generates a request and it is related to the *request producer* actor.

- Collecting Result. This part asks for a result and it is related to the *result consumer* actor.

The mentioned parts are important for the whole solution picture; see below for a more detailed description.

It is an environment for the following use cases of the following actors:

- Request Producer actor:
  - produce request
  - ask for confirmation
  - exchange result handler
- Result Consumer actor:
  - ask for result
  - stop producing result
  - exchange result handler

From the previous list it follows that the user component has to provide an exchanging and a result handler from the *generating result* part to the *collecting result* part.

Generating Request

A request can be generated basically in two ways:

- Human Interface. A human interface allows a user to fill out a form or set several controls. Then a user component can produce a human based request.

  This kind of interface has its special characteristics like error prone input, slow rated submission.

- Application Interface. An application interface is used by an application that requires an well defined set of commands that can be called[2].

  Its special characteristic is a fast rated submission.

Collecting Result

A result has to be collected in more steps. The engine component, that is the system actor, via the user interface provides a result in more parts than appear immediately when a source produced a part of the result.
This allows the immediate display of results to an end customer – to a human or machine readable form.

The collecting result component part has the possibility to stop the production of a requested result, that can be used only when a limited number of results is sufficient.

[2]Good implementation candidates are: Web Services, RMI, JMS, etc.

6.4.5 Administrator Interface

The administrator interface component is an interface between the engine and the administrator component. It handles the managing of scenarios and monitoring of the engine component.
It has to comply with the *monitoring* advanced requirements.

It is an environment for the following use cases of the following actors:

- Supervisor actor:
  - monitor system status
  - control scenario
  - deploy developed scenario

6.4.6 Administrator

The administrator component is a human interface component that uses features of the administrator interface component.

It provides tools for displaying and controlling scenarios, monitoring system status and deploying a developed scenario by a scenario developer in the developer component.

It is an environment for the following use cases of the following actors:

- Supervisor actor:
  - exchange developed scenario

Access to the administrator component has to be secured and therefore can also be provided for a remote administration.

6.4.7 Developer

The developer component provides a set of tools for developing a scenario, i.e. a set of sources with mapping to a result.

It has to comply with the *easy developing* advanced requirement.

It is an environment for the following use cases of the following actors:

- Supervisor actor:
  - exchange developed scenario
- Scenario Developer actor:
  - exchange developed scenario
  - develop scenario

An implementation of the developer component is not covered in this thesis.

6.4.8 System Actor Components

An example of how an actor can be split between components is shown in Figure 6.7 for the system actor.

The system actor is split between three components: *engine*, *user interface* and *manager*.

6.5 Components Design

A components design is an advance on the way to the implementation phase.

The first section shows a separation of components and their connections. An interaction between components is defined in the next section. The next sections present a design of selected components.

The result of these sections is a clear architecture overview as presented in Section 6.6.

6.5.1 Components Relationship

There can be two kinds of components:

- Client Component.
- Server Component.

A communication between two components is defined as a *connection*; and because there are two kinds of components, similar, three kinds of a connection are defined:

- Local Client Connection. An internal connection between client components.
- Local Server Connection. An internal connection between server components.
- Remote Connection. A connection between a client and a server component.

Each kind of connection has its specific characteristics – a *remote* connection can transfer less data with slower response times; a *local server* connection has to be reliable and fail-over; and for a *local client* connection there are no limitations.

The relationship between components, as shown in Figure 6.8, consists of three *client* components:

- user
- administrator
- developer

and the rest of components as *server* components:

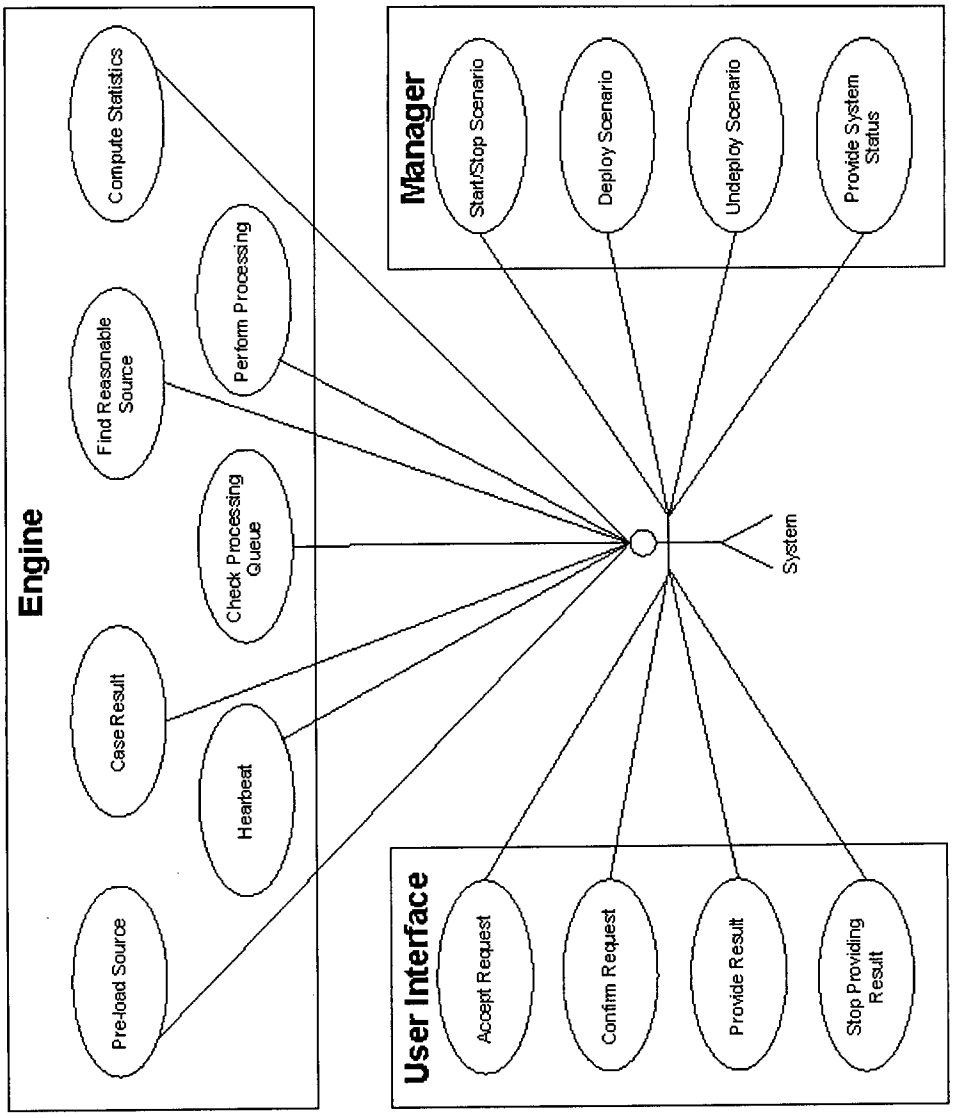


Figure 6.7: Components of the system actor.

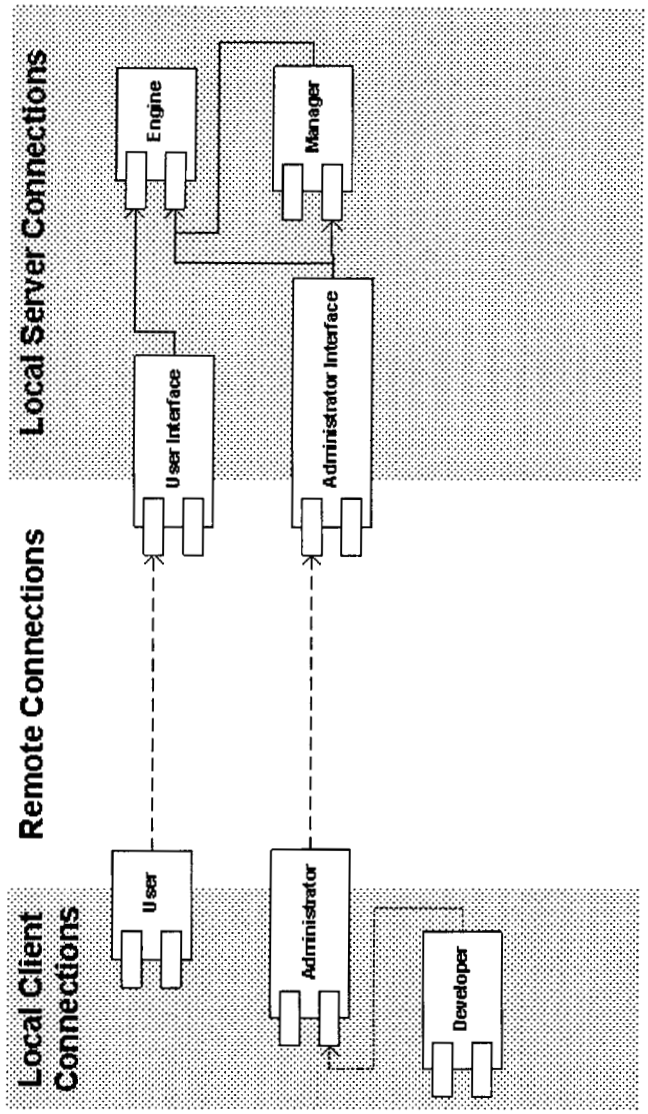


Figure 6.8: The separation of components and their connections.

- engine
- manager
- user interface
- administrator interface

The following connections emerge from the above-mentioned separation of components and it is shown in Figure 6.8:

- *local client* connections
  - administrator – developer
  - user (generating a request – collecting a result, i.e. request producer actor – result consumer actor).
- *local server* connections
  - engine – manager
  - engine – user interface
  - engine – administrator interface
  - manager – administrator interface
- *remote* connections
  - user interface – user
  - administrator interface – administrator

6.5.2 Components Intercommunication

The following diagrams of an intercommunication between the main components – *user*, *user interface* and *engine* – show possible flows in three cases:

- there is no cached result, see Figure 6.9.
- the first result is cached, see Figure 6.10.
- the providing of the second result is stopped, see Figure 6.11.

The *result 1* processing is a shorter time then the *result 2* processing.

6.5.3 Engine Design

The engine component is the most complex component – it has to connect two different technology worlds: a reliable and distributed server with an Internet browser. This results in two parts of this component: core part and Internet data processing part.

Core Part

The core part is deployed on an application server that provides a distributed and reliable environment. Functionality is divided into logical units like tasks, data stores. An application server starts tasks as necessary and is able to start a task on another application server in a cluster if a load is higher than a defined threshold.

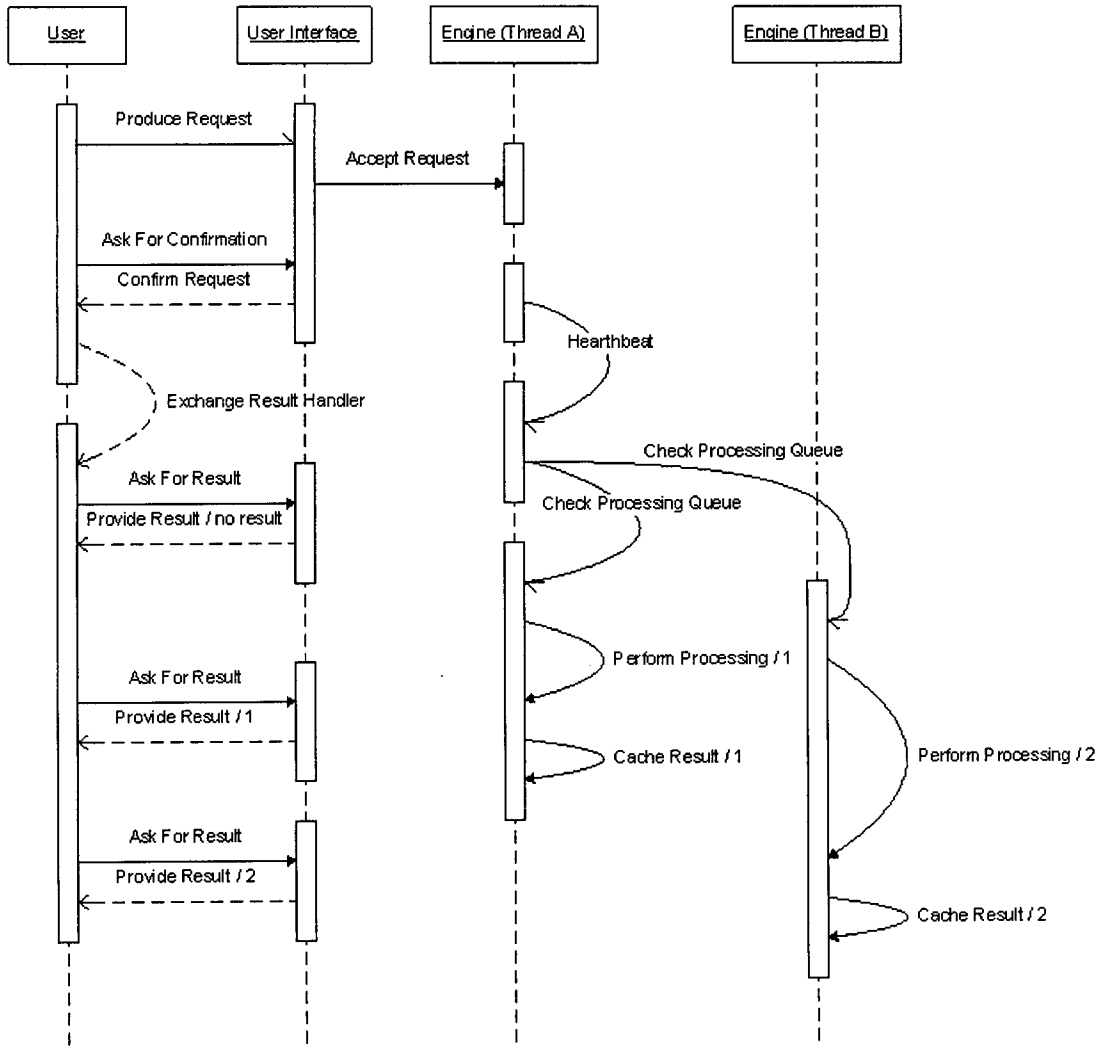


Figure 6.9: An intercommunication between components with no cached result.

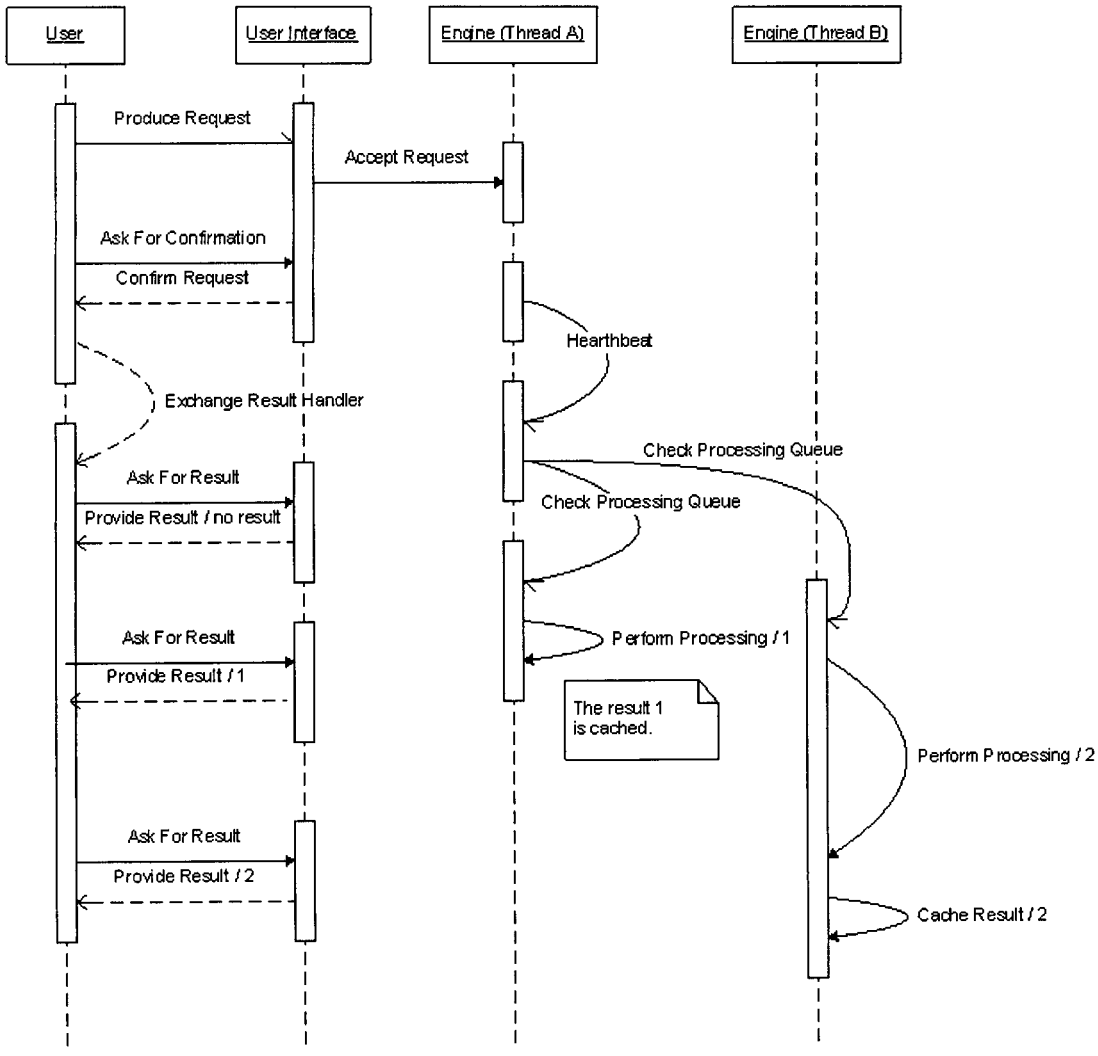


Figure 6.10: An intercommunication between components with a cached result.

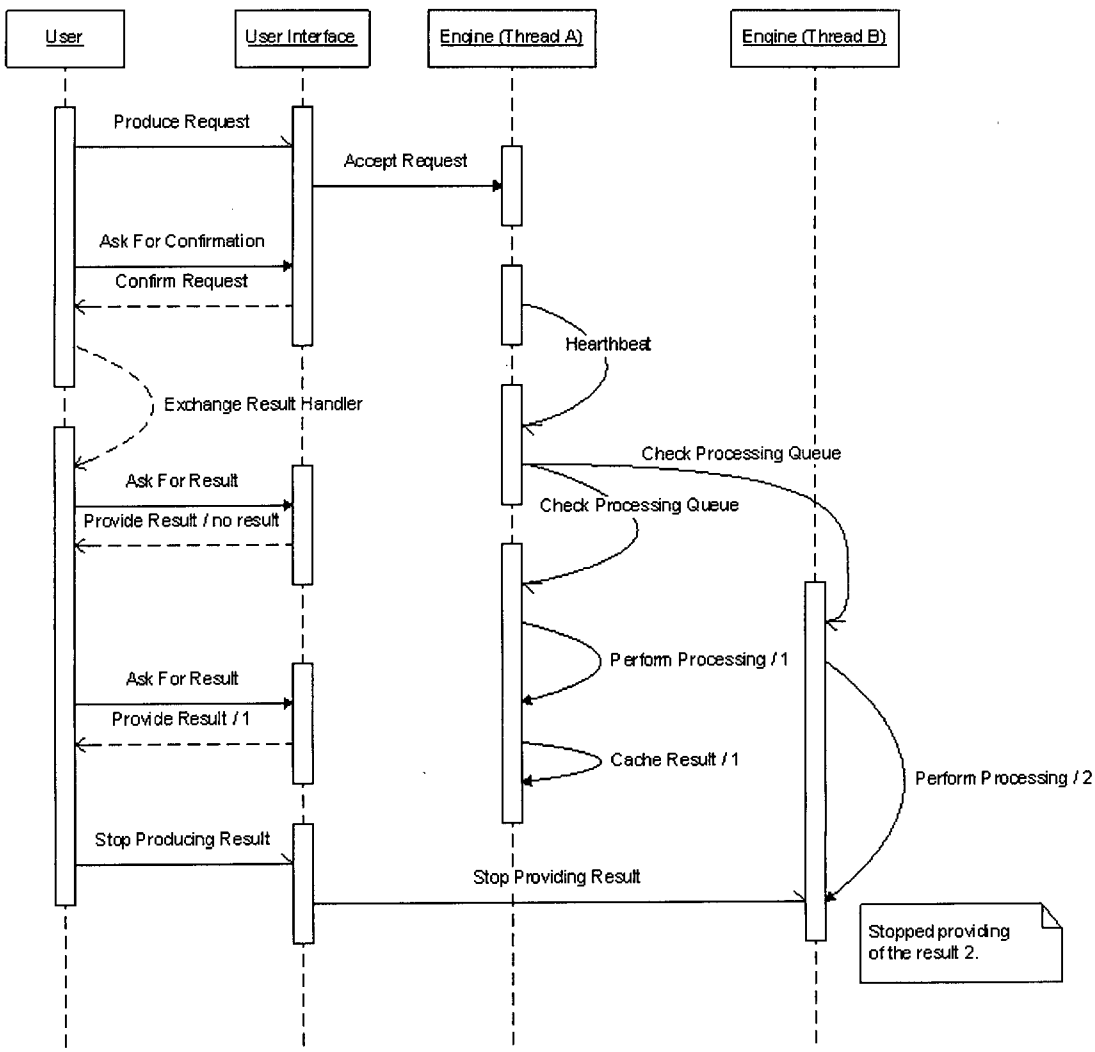


Figure 6.11: An intercommunication between components where the processing and providing of a result was stopped.

Internet Data Processing Part

The most appropriate environment for the Internet data processing part is a browser. A browser can handle the majority of Internet standards and technologies such as HTTPS, cookies, JavaScript.

The solution, as a part of the engine component, has to be reliable, a direct connection to a browser can not be used – it is necessary to *wrap* it.

A wrapped browser should also provide an extraction functionality, i.e. extracting data from an Internet source like creating an XML document from a web HTML page.

The engine components are also capable of processing more sources parallel, that is, it is necessary to start more then one wrapped browser instance.

Additionally, the technologies used to wrap a browser, so it can be used by the engine component, are incompatible with technologies used for the core of the engine component. For more information see Section 7.3.

The solution inhere in creating two separate segments:

- Downloader & Extractor. Basically it is a wrapped browser with an extraction module. It is capable of navigating to a source, downloading it and extracting required data from it.

  An integration of a wrapped browser and an extraction module connects two logically and technically close elements. Logically, because the extraction step immediately follows the navigation and downloading step. Technically, because an extractor module can work on the downloaded document directly.

  For more technical detail, see Section 4.4.3 and Section 7.3.

  The possibility to *parameterise* a navigation is a feature provided by the downloader that is essential for a metasearch solution.

  All user inputs such as typing or moving with a mouse can be parametrised, i.e. their values[3] can be marked as parameterisable and their values can be defined externally. An external process such as a metasearch engine or Lixto Transformation Server specifies values for parametrised user inputs in a navigation. The navigation is then executed with changed values, e.g. a user fills out a city name input field – *London* is typed. The typed sequence is marked as parameterisable. Metasearch loads the navigation and changes the value of the parametrised input to *Vienna*. The navigation is executed as if a user had typed *Vienna* there.

- Dispatcher. It dispatches incoming requests from the core part to browser instances. Browser instances are created by the dispatcher.

  The dispatcher sends a request from the engine part to a not busy instance. If an instance does not respond till a threshold, the instance is terminated and the request is re-sent to a non busy or a new instance.

[3] For the typing input a value of the pressed keys and the number of pressed keys can be changed. For the mouse input a position of the recorded points can be changed.

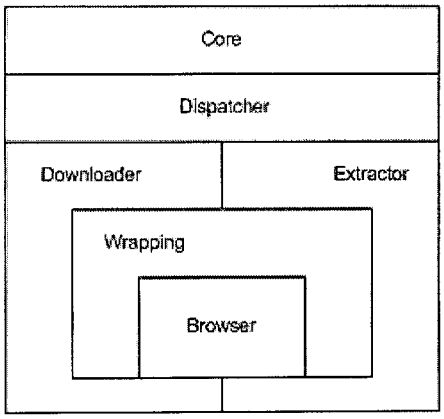


Figure 6.12: A schema of the engine component. The browser element functionality is wrapped. The downloader and extractor elements access the wrapped browser, but also have a direct access to the browser element. The dispatcher element provides an abstraction layer for the core element.

The dispatcher segment creates an abstraction layer between the core part and the downloader & extractor segment. The core part does not have to solve problems like a not responding browser instance.

The abstraction also brings the possibility to use another browser or system, or to use more systems – the core part does not have to be changed.

For a schema see Figure 6.12.

6.6 Architecture

The architecture design is the last step in the design of the whole solution. In the previous sections everything was prepared for it.

The implementation phase, in the next chapter, is built on the architecture design.

A simplified design overview, see Figure 6.13, shows basic components user, administrator and engine.

It shows fundamental components and connections between them.

A complex design overview, see Figure 6.14, extends the simple overview with elements described in Section 6.5.3 and a new element *data storage*:

- User
- Administrator
- Engine
- Dispatcher
- Downloader & Extractor

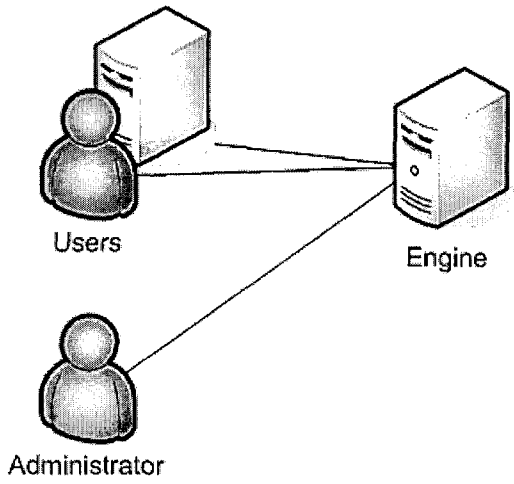


Figure 6.13: A simplified architecture design overview.

- Data Storage

The *data storage* element is added as a consequence of various needs from use cases and component designs, for example a process queue has to be stored in a data storage to facilitate reliability.

It has to comply to general requirements for a database, like it has to support SQL and transactions.

The next Chapter *Implementation* connects the design with technologies to meet requirements.

6.7 Interconnection with Transformation Server Design

A design of the new Lixto Transformation Server product version is interconnected with the design presented in the previous sections. Parts of the metasearch design can be reused completely or partially, but in general the architecture is the same.

The engine - dispatcher intercommunication is also the same for TS. This part can be reused completely.

The request - response model allows simplification of the metasearch engine component – there is no need to start user processes at specified times, there are only predefined system tasks such as caching. The TS engine component will be more complex, but it can reuse issues such as accepting requests, providing results, caching, heartbeat and technical areas such as data storage access.

The administration functionality is similar for both products, although TS has to provide more information, because it is a more complex system, the way of providing them is the same. This part can be reused completely.

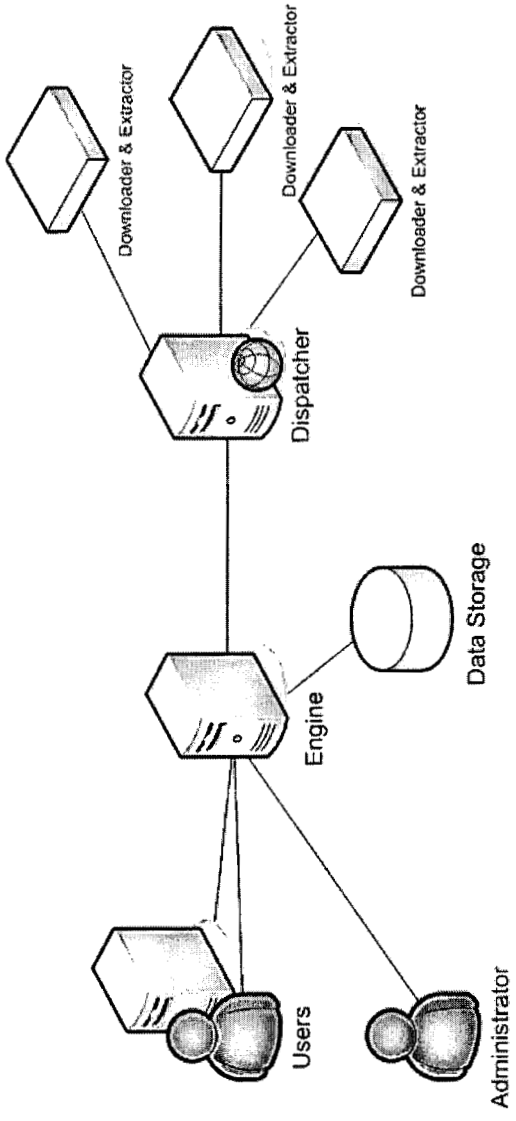


Figure 6.14: A complex architecture design overview.

The client component and the intercommunication with a client or another system and the server are different and they have other requirements. Only technically related issues such as a client - server communication can be reused.

This thesis does not only create a new metasearch solution design, but also lays the fundamentals of a new version of the Lixto Transformation Server product and introduces appropriate technologies and their application.

Chapter 7

Prototype Implementation

A prototype implementation of the Snorri metasearch solution designed and developed in this thesis is shown in this chapter. It is a complex system that uses advanced enterprise technologies.

The first section *Technologies and Application* briefly explains the technologies used and their application in the implementation.
The second section *Overview* describes the implementation and the subsequent sections such as *Server* and *User* show interesting implementation details.

7.1 Technologies and Their Application

The technologies that are used in the implementation are briefly described in the following section, where the application of each mentioned technology in the implementation is specified.
For each mentioned technology its application in the implementation is described.

7.1.1 Java

The Java programming language is a general-purpose, concurrent, class-based, object-oriented language that is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted. It is intended to be a production language, not a research language. It was developed by Sun in 1990s.
The Java programming language is strongly typed and its specification clearly distinguishes between the compile-time errors that can and must be detected at compile time, and those that occur at run time. Compile time normally consists of translation of programs into a machine-independent byte code representation. Run-time activities include loading and linking of the classes needed for execution of a program, optional machine code generation and dynamic optimization of the program, and actual program execution.
A compiled machine-independent byte code is executed in a platform-dependent environment called Java Virtual Machine (JVM).

The specification [Sun05a] says: *"The Java programming language is a relatively high-level language, in that details of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation (as in C's free or C++'s delete). High-performance garbage-collected implementations can have bounded pauses to support systems programming and real-time applications. The language does not include any unsafe constructs, such as array accesses without index checking, since such unsafe constructs would cause a program to behave in an unspecified way."*

For the implementation purposes the Java programming language is an easy-to-use, safe and platform-independent programming language with a rich set of various libraries, such as regular expression, HTTP support. One of the main advantages is the enterprise edition described in the next section.

7.1.2 Java Enterprise Edition

The Java Enterprise Edition, *Java EE* or previously *J2EE*, is a set of specifications listed in the [Sun05c] specification – Web Service, JSF, JSTL, EJB 3.0, JAXB and JAX-WS – for development and execution of a multi tier application.

The required relationships of architectural elements of the J2EE platform are shown in Figure 7.1. The figure shows the logical relationships of the elements.

The [Sun05c] specification describes the above mentioned figure as follows: *"The Containers, denoted by the separate rectangles, are J2EE runtime environments that provide required services to the application components represented in the upper half of the rectangle. The services provided are denoted by the boxes in the lower half of the rectangle. For example, the Application Client Container provides Java Message Service (JMS) APIs to Application Clients, as well as the other services represented.*

The arrows represent required access to other parts of the J2EE platform. The Application Client Container provides Application Clients with direct access to the J2EE required Database through the Java API for connectivity with database systems, the JDBCTM API. Similar access to databases is provided to JSP pages and servlets by the Web Container, and to enterprise beans by the EJB Container."

For a metasearch implementation the Java EE technologies provide an industrial standard for creating distributed multi tier applications, i.e. the implementation has features such as fail-over, scalability and distribution of load.

The following sections describe the main and most widely used Java EE technologies and services.

7.1.3 Web Services

The Web Service standard describes a communication technology between two processes mainly over a network. Both sides of communication can exchange a set of XML documents or use a defined structure of a communication protocol such as SOAP, where a request contains a specific structure to call a remote method on the other hand, and a response can contain result data or an error structure with the error description.

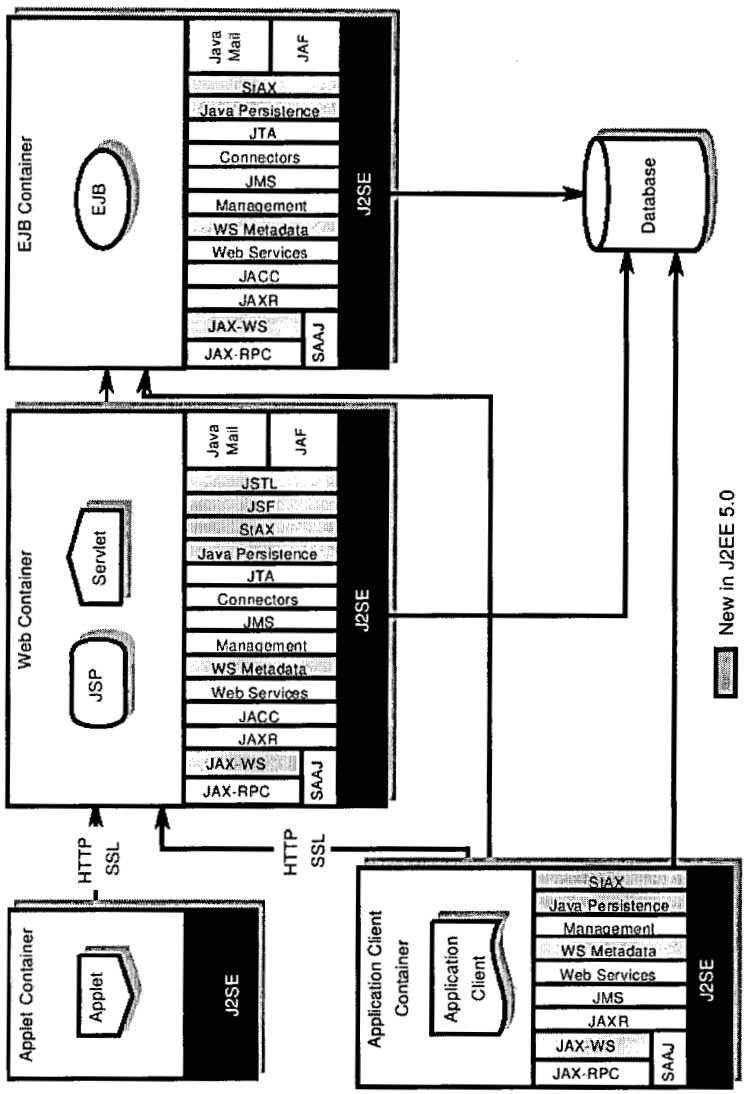


Figure 7.1: J2EE architecture [Sun05c] diagram.

A web service is described by Web Service Description Language, WSDL. A descriptor contains all necessary data to start communication.

The [Con04b] specification says: *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."*

For the metasearch implementation the web service standard and library provide an industrial standard for the exchange of data between other systems, such as clients that produce requests, and systems that receive results.

Technically the metasearch solution contains a web service for receiving requests and a web service for providing obtained results. The use of web services for intercommunication with other systems facilitates the implementation of a producer of requests and a system for the processing of results independently of the metasearch implementation, such as a programming language, environment and hardware platform.

Web Service standard and its technologies such as XML, SOAP and WSDL represent a commonly used implementation in Service-Oriented Architecture, SOA – that is a system design methodology for the creation of reusable business processes.

The description language *WSDL* and the protocol *SOAP* are described below.

Web Services Description Language

Web Services Description Language version 2.0, WSDL 2.0, provides an XML format for describing web services. WSDL 2.0 allows to separate the description of an abstract functionality from concrete details of a service description, such as *how* and *where* that functionality is offered.

The [Con06b] specification says: *"WSDL 2.0 describes a Web service in two fundamental stages: one abstract and one concrete. Within each stage, the description uses a number of constructs to promote reusability of the description and to separate independent design concerns."*

At an abstract level it describes a web service as it sends and receives messages. Messages are described independently using a type system such as XML Schema. Operation associates a message exchange pattern with one or more messages. A message exchange pattern identifies the sequence and cardinality of messages sent and/or received. Interface groups operate without any commitment to transport.

At a concrete level a binding specifies transport and wire format details for one or more interfaces. An endpoint associates a network address with a binding and a service groups together endpoints that implement a common interface.

Simple Object Access Protocol

Simple Object Access Protocol, *SOAP*, is a communication protocol based on XML.

The [Con06a] specification says: *"SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics."*

The protocol provides a way to encapsulate method calls with parameter types and values, returning types and exceptions.

7.1.4 Java Message Service

The Java Message Service, *JMS*, provides a method to exchange messages via a messaging service for Java applications. This allows a separation of different business components into a reliable and flexible system.

The communication between components is transmitted through a Message Oriented Middleware product, *MOM*. It facilitates the receipt of sent messages and their distribution to specified receivers. A message can be delivered reliably, i.e. stored and delivered when the receiver will be able to receive it, e.g. it recovers from a crash.

The [Sun01] specification says: *"JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product."*

For the metasearch implementation the JMS technology provides a reliable means of communication between business and logical parts of the solution.

7.1.5 Java Server Pages

The Java Server Pages, *JSP*, platform is an Java Enterprise Edition technology for the creation of applications with dynamically generated web content. It simply enables generation of a content for web standards, such as HTML, DHTML, XHTML and XML.

The [Sun06b] specification says: *"JSP technology provides the means for textual specification of the creation of a dynamic response to a request."* and *"A JSP page is a text-based document that describes how to process a request to create a response. The description intermixes template data with dynamic actions."*

The technology is built on the concepts of template data, where templates are text or XML fragments, on addition of dynamic data to templates and encapsulation of functionality, where JavaBeans component architecture facilitates an access to data and tag libraries provide custom actions, functions and validation.

For the implementation the JSP pages represent the basis for the creation of web based graphical interfaces, such as the administrator console; for more details about the implementation of the administrator console of Snorri refer to Section 7.5.

7.1.6 Java Server Faces

Java Server Faces, *JSF*, is a user interface, *UI*, framework for web applications written in Java. As described in the [Sun06a] specification, it simplifies design, writing and maintenance of user interfaces by easy creation and reuse UI components, simplification of migration of data between an application and UI, and by providing a simple model for wiring events produced by a client to server application code.

JSF is protocol and markup language independent. However, as described in the [Sun06a] specification, it helps to solve many common problems with writing an application for HTML clients that communicate via HTTP to a Java application server with the support for JSP based applications. It supports various form processing methods such as a multi-page form, validation of a request, error handling with reporting in a human-readable form, and type conversion.

For the metasearch implementation it is a matter of course to choose the application of a Model-View-Control, *MVC*, framework with the support of JSP as a template technology. JSF is used for the creation of the administrator console UI; for more details see Section 7.5.

7.1.7 Enterprise Java Beans

The Enterprise Java Bean, *EJB*, technology is one of the Java EE technologies. It is a complex technology for the creation of enterprise applications, for more details see the [Sun05b] specification.

The implementation utilities the latest version 3.0 that noticeably simplifies the whole development process in that is uses Java annotations and the dependency injection, contains suitable components such as EJB Timer, that can define a specific time when it should be activated and perform a defined task, and solves serious drawbacks of the previous versions, such as creating a primary key.

One of the main simplifications is the use of Java annotations that removed one of the most complex part of the development phase – creation and maintenance of description files.

Each enterprise bean represents a business logic element – a programming artifact. Each bean consists of an interface and an implementation class. The necessary features are annotated using Java annotations.

For different purposes various types of enterprise beans are defined. They are described in the following sections.

Stateless Session Bean

A stateless session bean represents an action in a business model. It does not contain a state, that is, every call has to take all its parameters; the advantage of this is easier distribution of such calls in a cluster as for the stateful session beans.

For a life cycle see Figure 7.2.

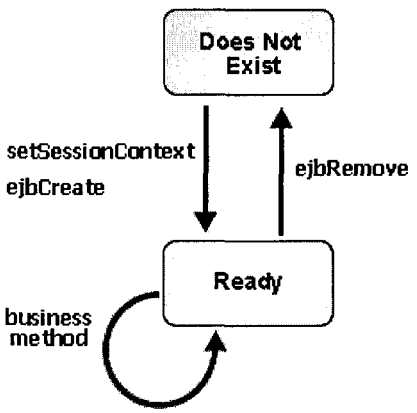


Figure 7.2: Simplified stateless session bean life cycle [BEA03] diagram.

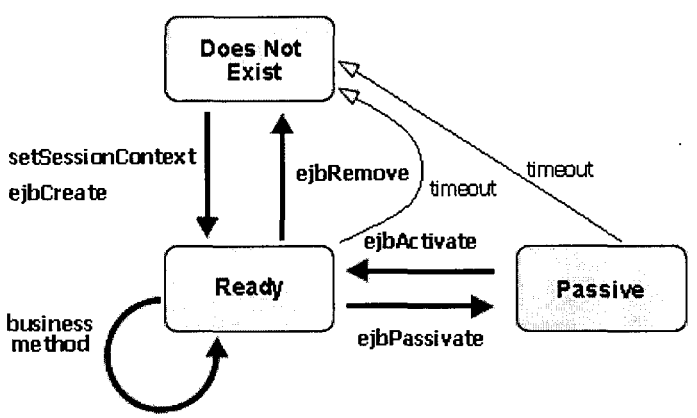


Figure 7.3: Simplified stateful session bean life cycle [BEA03] diagram.

Stateful Session Bean

A stateful session bean, like the stateless version, represents an action, but it saves its state between calls.

In practise, stateful session beans are not used and a lot of literature, such as the [RAJ02] book, recommend not using them. An alternative to them is a combination of a stateless session bean and an entity bean or just a stateless session bean with all parameters on the input.

For a life cycle see Figure 7.3.

Entity Bean

An entity bean represents a business data object that is persisted. The specification describes necessary features for the persisting functionality such as transactions and isolation levels.

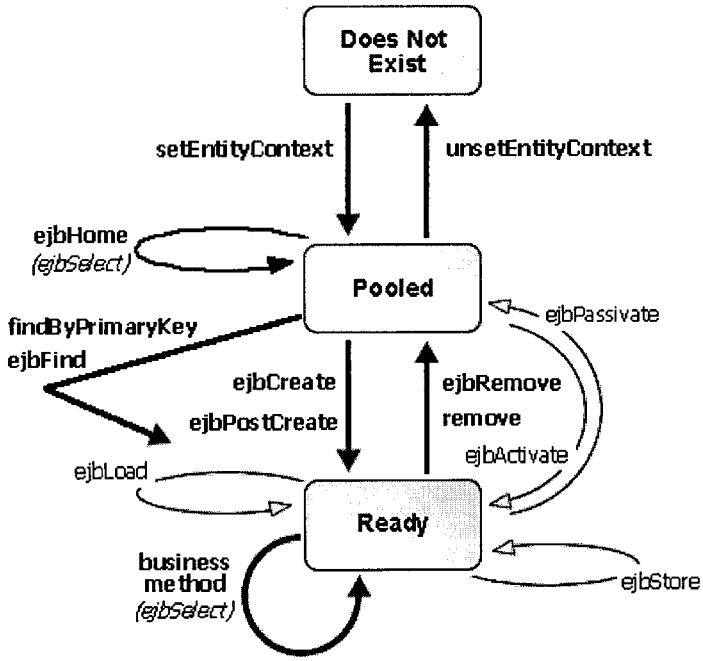


Figure 7.4: Simplified entity bean life cycle [BEA03] diagram.

The development and use of entity beans in EJB 3.0 is significantly simplified compared to the previous versions. Main functions, such as the creation of a new instance of an entity bean, are available in the entity manager.

It is recommended using session beans, rather than calling an entity bean directly – it naturally corresponds with the reality, where actions modify a state of data objects.

For a life cycle see Figure 7.4.

Enterprise JavaBeans Query Language

The Enterprise JavaBeans Query Language, *EJBQL*, is a query language based on SQL92. It is used for the definition of finder and select queries for an entity bean, that is a possibility exists to search for an entity bean instance rather then to search for a row in a common relational database.

The new EJB version 3.0 brings features in EJBQL, such as grouping and sorting, that improve the usability.

Message-Driven Bean

The last type of enterprise beans, the message-driven bean, *MDB*, is a specialised type designed to receive a JMS message. It simplifies the process of receiving a message and propagating its content to other entity bean, such as a stateless session bean.

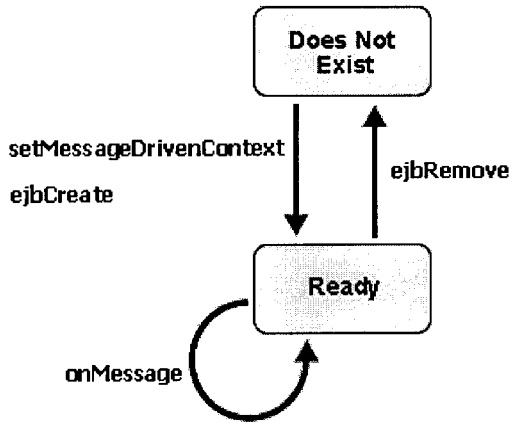


Figure 7.5: Simplified message-driven bean life cycle [BEA03] diagram.

Message-driven beans represents the only way in the EJB world to split a process into more subprocesses, that is to emulate starting a new thread in the Java Standard Edition. This feature is used in the metasearch implementation to start a set of processes for a new request handling.

For a life cycle see Figure 7.5.

7.1.8 Java Native Interface

The Java Native Interface, *JNI*, allows the Java code to call applications and libraries written in other programming languages, such as C++.

The [Sun03a] specification illustrates possible usage:

- The standard Java class library does not support the platform-dependent features needed by the application.
- You already have a library written in another language, and wish to make it accessible to the Java code through the JNI.
- You want to implement a small portion of a time-critical code in a lower-level language, such as the Assembly language.

In the metasearch implementation the JNI is used for communication with the Mozilla browser; for more details see Section 7.3.4.

7.1.9 Databases

The [Wik06] Wikipedia page says: *"A database is an organized collection of data."*

The metasearch solution uses a database as storage for its configuration settings, scenario settings, such as form descriptors and mappings, and run-time data, such as cached results.

In the implementation only the Relational Database Management Systems, *RDBMS*, are considered.

There are commercial database engines such as Oracle DB, IBM DB2, or open-source database engines, such as PostgreSQL and MySQL.

The PostgreSQL is used in the Snorri metasearch prototype implementation for its features and the open-source character. Since PostgreSQL is similar to Oracle DB, in case of increased requirements the Oracle DB can be used with minimal costs of migration.

7.1.10 Application Servers

An application server is a component-based product that resides in the middle-tier of a server centric architecture. It provides middleware services for security and state maintenance, along with data access and persistence.

The most comprehensive and the latest list of J2EE application servers is available on the *Application Server Matrix* [The05] page. Another page [Jav06a] contains a list of Open Source EJB Servers.

Basically, there are only two application servers that support the EJB 3.0 standard: JBoss Application Server and Oracle Application Server.

JBoss Application Server

At present the server that is most widely used and accepted for the *EJB3* architecture is *JBoss* described on the [JBo06] page. It is open-source software with large community and support.

The Snorri metasearch prototype implementation uses this application server for its features and the open-source character.

Oracle Application Server

As a commercial alternative, the Oracle Application Server, *Oracle AS*, is described on the [Ora06] page.

7.2 Overview

The Snorri metasearch prototype implementation is described in the following sections.

The prototype implementation represents a vertical prototype that implements crucial parts of the solution to point out any insufficient design or technology part. In the vertical prototype some functionalities and features are simplified or simulated.

The architecture implementation overview, see Figure 7.6, shows assigned implementation solutions to the elements described in Section 6.6 and pictured in Figure 6.5.3:

- The user element is implemented as a set of JSP pages, for the graphical user interface access suitable for a human user, and as a set of web services, that are designed for the application – application communication. For details see Section 7.4.

- The administrator element is implemented as a JSP page. For details see Section 7.5.
- The implementation of the engine element is represented by the Snorri server. For details see Section 7.3.1.
- The dispatcher element is implemented as the Octopus server. For details see Section 7.3.3.
- The implementation of the downloader & extractor element is represented by the Lixto Navigator and Visual Wrapper products. For details see Section 7.3.4.
- The PostgreSQL database stands for the data storage element. For details See 7.3.2.

7.3 Server

The server implementation is based on the above mentioned Java EE technologies, mainly EJB 3.0. The main part, the engine, uses several subsystems to provide the designed functionality – a database instance for a reliable storage of data, Octopus for the evaluation of JavaScript scripts and calling Lixto Navigator and VW, and the mapping framework implementation prototype for the mapping of global parameters and values to local.

7.3.1 Engine

The engine prototype implementation, for a class diagram see Figure 7.7, has the following enterprise beans:

- The *Input* entity bean stores data about a request. The parameter – value pairs are stored in the *ParameterValue* entity bean.
- The *Request* stateless session bean is a gateway for user requests. It stores the received requests in the *Input* and *ParameterValue* entity beans and returns the id of the Input entity bean as the handler.
- The *Heartbeat* stateless session bean is the core bean of the system. It also is an EJB Timer where a method tagged with the *Timeout* annotation is called every defined interval; during call the *Input* bean is checked for a new unprocessed input. If there is an unprocessed input, the processing flag is set to true, the input as a set of global parameters and values is mapped to local parameters and values that are then sent via the Octopus server to Lixto Navigator and VW for downloading and extracting data from local search engines. The received data are stored in the *SearchResult* entity bean.

  The timer has to be started automatically; this is done in the *management* JSP page that is loaded automatically and in its constructor calls the *start* method of the *Heartbeat* bean.
- The *Response* stateless session bean is a gateway for providing results. It checks if there are unsent results for the specified handler.

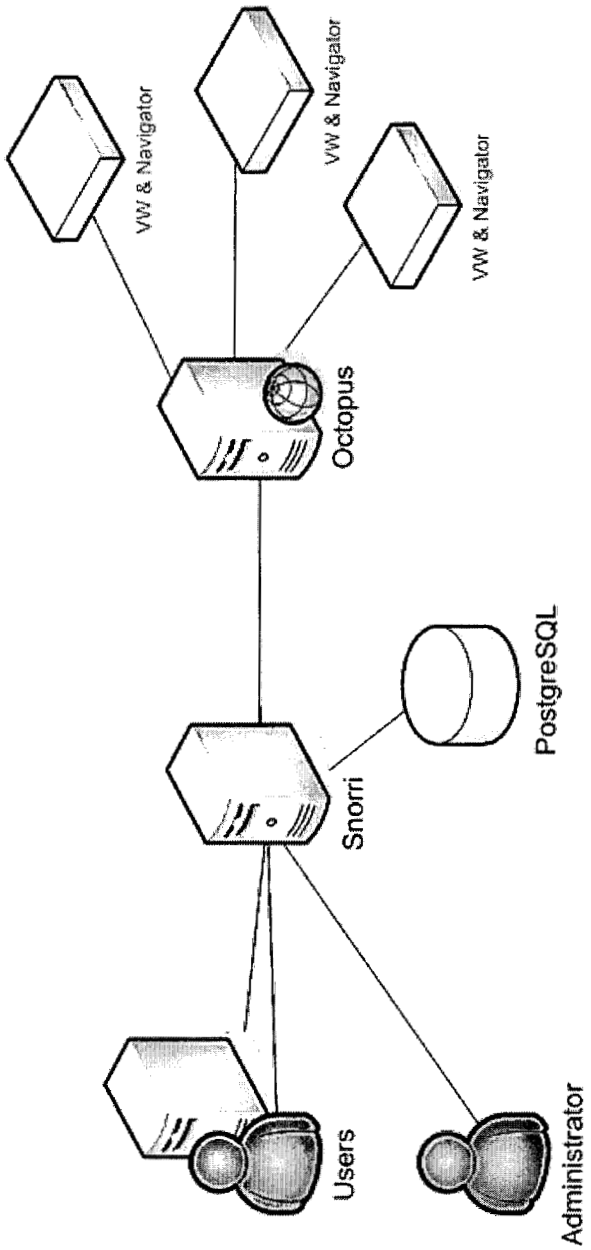


Figure 7.6: Architecture implementation overview.

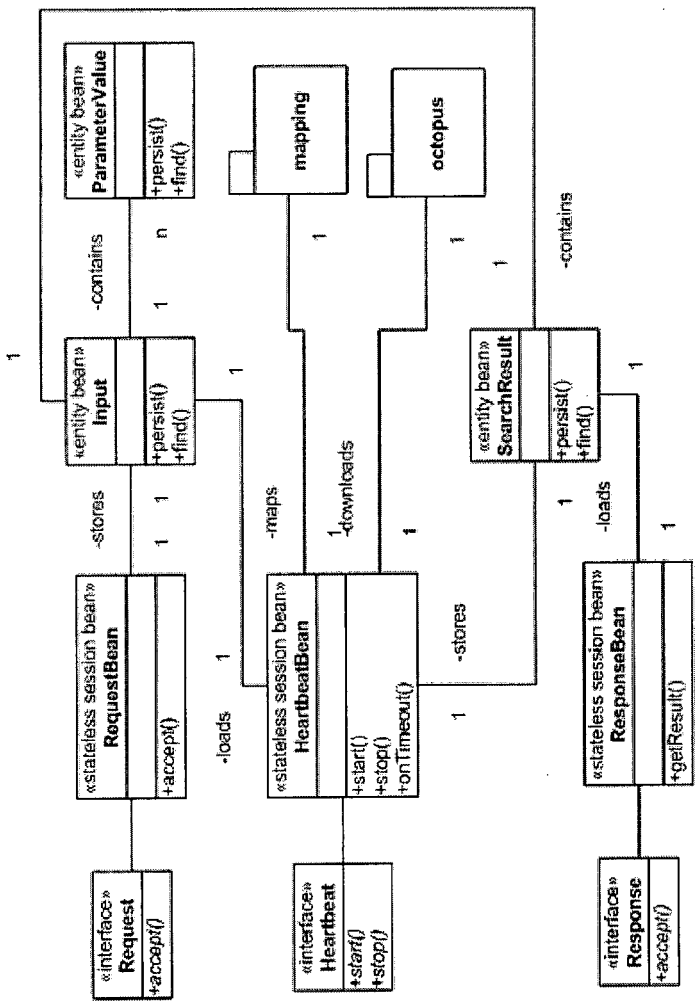


Figure 7.7: Engine prototype implementation class diagram.

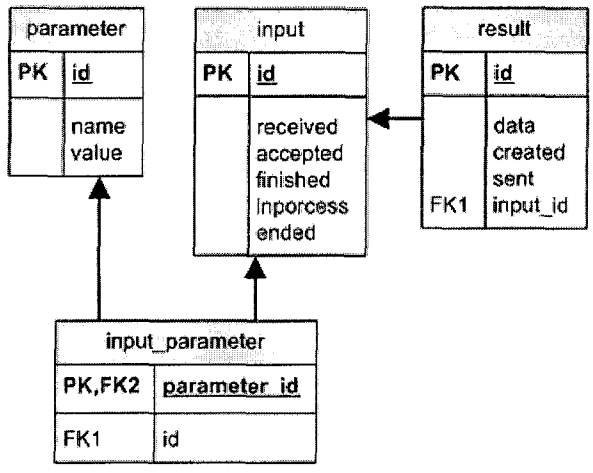


Figure 7.8: Database entity relationship diagram.

Appendix B on page 150 shows some notable parts of the source: the *Input* entity bean in Figure B.1, the *ParameterValue* entity bean in Figure B.2, the *SearchResult* entity bean in Figure B.3, the *Request* session bean in Figure B.4, the *Heartbeat* session bean in Figure B.5, and the *Response* session bean in Figure B.6.

The engine is deployed in the JBoss Application Server.

The JBoss Application Server is a reliable Java EE application server and hence it is used in the prototype implementation.

The engine prototype implementation fulfils the accessing intermediate results requirement and theoretically fulfils the requirements for scalability and reliability that are guaranteed by the used technologies, although the implementation itself was not tested for a higher load. It solves all main technical problems and it represents a proof of the correct design concept.

7.3.2 Database

Table structures in the PostgreSQL database are automatically created by the JBoss application server from the entity beans; for a diagram see Figure 7.8.

The *Input* entity bean is mapped to the *input* table, the *ParameterValue* entity bean to the *parameters* table, the $1 : n$ relationship between them is stored in the *input_parameter* table, and the *SearchResult* entity bean is stored in the *results* table.

The usage of a database such as PostgreSQL in the prototype implementation fulfil the requirements for reliability.

7.3.3 Octopus

The Java EE technologies bring advantages, such as a robustness, but also disadvantages mentioned in the previous chapters, such as disallowed usage of features such as JNI, threads or I/O operations. That is the reason for the

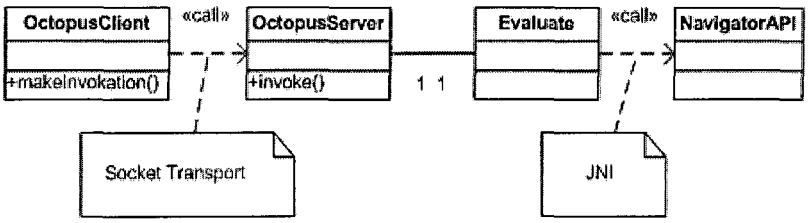


Figure 7.9: Octopus package prototype implementation class diagram that focuses on the communication between the engine and the Lixto Navigator & Extractor.

development of the Octopus server – a small isolated functionality accessible via allowed techniques, such as RMI.

The Octopus prototype implementation communicates with the engine via the JBoss Remoting library that provides a simple-to-use functionality for communication between two processes. It allows to use several communication methods, such as a simple socket connection or RMI. The prototype implementation uses the socket connection between communication points.

The Octopus server prototype implementation solves the following problematic areas:

1. *Communication* with Lixto Navigator and Extractor, i.e. Octopus replaces Hydra, for more details see Section 4.4.3. The main reasons are: simplification of the prototype implementation – for the deployment process, because JBoss Remoting is included in the JBoss AS, and for the implementation process, because it represents a high-level communication library.

2. Evaluation of JavaScript. JavaScript is a scripting language of the mapping framework prototype implementation; for more details see Section 5.10. In other words, Octopus provides a connection between the engine and the Rhino scripting library.

For a class diagram of the Octopus prototype implementation see Figure 7.9. The Octopus prototype implementation solves all related technical problems.

7.3.4 Navigator & Data Extractor

The Lixto Navigator and Data Extractor product as a part of the Lixto Visual Developer is accessed via its *NavigatorAPI* interface.

In the prototype implementation it is started manually. A production implementation has to be able to start and stop Lixto Navigator and Extractor instances automatically.

For more details about the Lixto Navigator and Extractor see Section 4.2.

Using the Lixto Navigator and Extractor helps to meet the faster development, session identification assignment and multi-page search forms requirements.

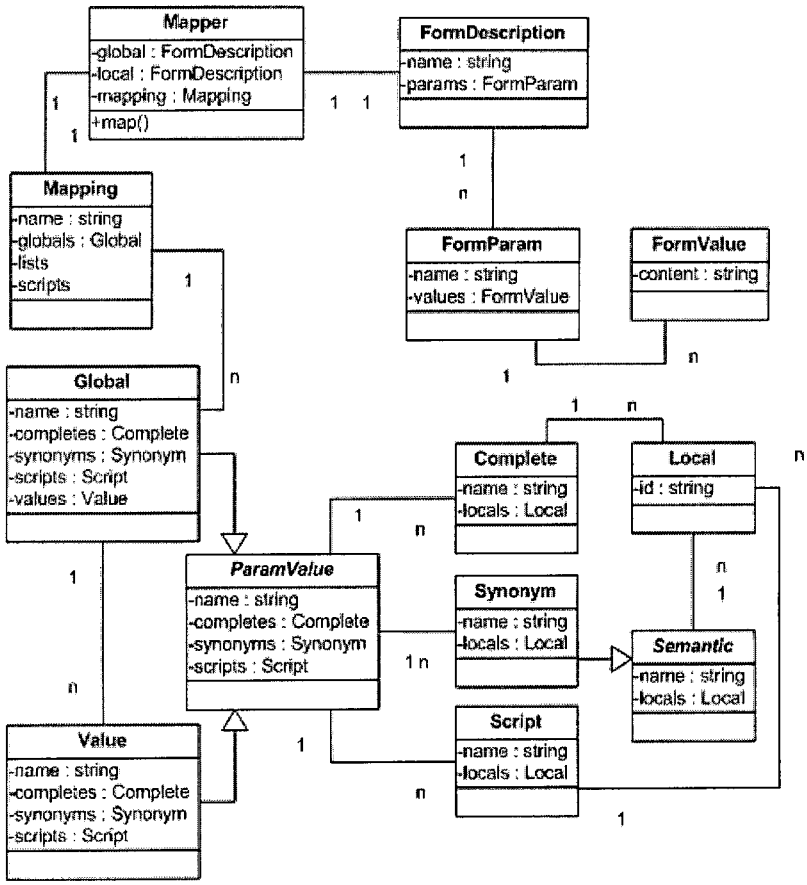


Figure 7.10: Mapping package prototype implementation class diagram.

7.3.5 Mapping Framework

The mapping framework prototype implementation covers all main parts: global and local form descriptors; complex, synonyms, combined and script mapping notation with the list mapping support. For more details see Chapter 5.

See Figure 7.10 for the class diagram of the mapping framework prototype implementation.

The mapping framework prototype implementation is incorporated in the engine. The engine calls directly the main gateway *Mapper* class that accepts global and local form descriptors, a mapping, and a set of global parameter – value pairs and returns a set of parameter – value pairs for each local search form.

The prototype implementation of the mapping framework fulfils the faster development requirement. It implements the main parts of the mapping framework design, solves all technical problems and represents a solid base for a mapping framework with all designed features.

7.4 User

The user element prototype implementation covers the following two areas; for a class diagram see Figure 7.11:

1. It is an exchange point for graphical user interfaces suitable for a human user. The implementation consists of two JSP pages: *search.jsp* for the receipt of requests and returning of a handler and *result.jsp* for the provision of results.

   The *search.jsp* calls the *Search* stateless session bean that represents a simplified engine-user interface; for more details see 6.4. The bean calls the engine's *Request* bean. The JSP page generates an XML response with a handler.

   The *result.jsp* calls the *Response* stateless session bean that calls the engine's *Response* bean. The JSP page generates an XML response with result data.

   For the source code of the *result.jsp* page see Figure B.7 on page 153.

2. It represents a gateway for the application – application communication. The implementation consists of two web services: *Search* and *Result*. Similar to the previous *human interface*, the *application interface* calls the engine's *Request* and *Response* beans.

   Technically EJB 3.0 allows to tag a stateless session bean with special annotations that allow to use it as a web service. The source code of the *Result* web service consists of two parts: an interface, shown in Figure B.8 on page 154, and an implementation as a bean, shown in Figure B.9 on page 154.

The prototype implementation covers the scalability and accessing intermediate results requirements. It solves all technical problems.

For a prototype of a user interface with dynamic loading of results see Chapter 8.

7.5 Administrator

The administrator prototype implementation is simplified to one JSP page, the *management.jsp* page, that calls directly the engine's *Heartbeat* bean and allows to start or stop the timer, i.e. to manage the processing of received requests.

For a class diagram see Figure 7.11.

For a page see Figure 7.12.

The administrator prototype implementation partially fulfils the monitoring requirement. It provides a simple base, where main technical problems are solved, and it can be easily extended to provide more information and operations.

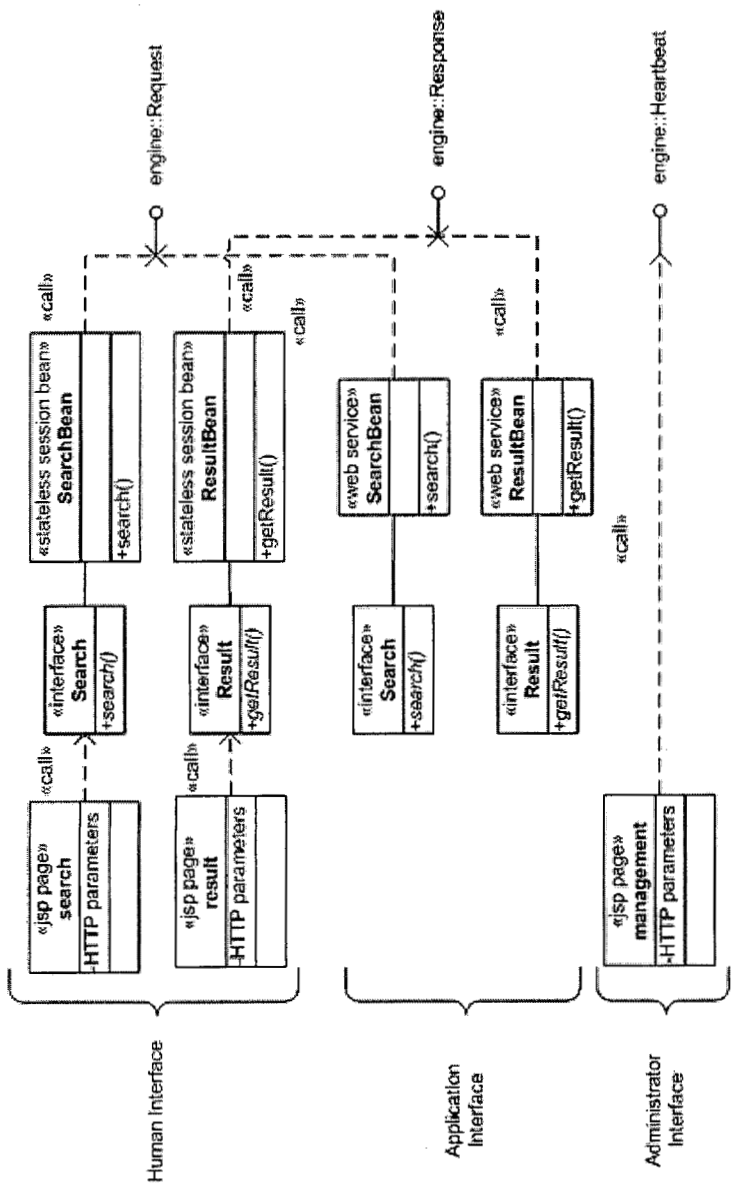


Figure 7.11: User and the admin package prototype implementation class diagram.

Management console

Operation start hearbeat process finished successfully.

Heartbeat process	running	start	stop
Accepting requests	active	activate	deativate
Heartbeat interval	1 second	s	set

Figure 7.12: Administrator management console page. The start operation represent the last operation.

7.6 Interaction

The interaction of the implemented elements mentioned above is shown in Figure 7.13. It is extended to include elements from the following Chapter 8 – the *flight_search.jsp* search page and the *flight_result.jsp* result pages that are described in Section 8.6 – so that the sequence diagram can show all interactions.

The interaction covers the whole request – response process, i.e.:

- production, sending and receipt of requests,
- creation and return of a handler,
- loading and mapping of input data,
- calling the downloading and extracting functionality,
- storage of results,
- loading and return of results.

Figure 7.14 shows a different on the interaction. It describes data flow between the implemented elements:

1. A user enters data on the global search form.
2. The user submits data that are transformed to a request.
3. A request is sent to the server, to the *RequestBean* stateless session bean.
4. The *RequestBean* stateless session bean stores the received request in the *Input* entity bean, that is persisted to the database. The persisted *Input* entity bean gets new unique identification, an *ID*, that is used as a handler in the subsequent steps.
5. A result page is generated with the new handler.
6. The *Heartbeat* stateless session bean with an EJB Timer is periodically called. It checks if there are any new unprocessed *Input* instances in the database.

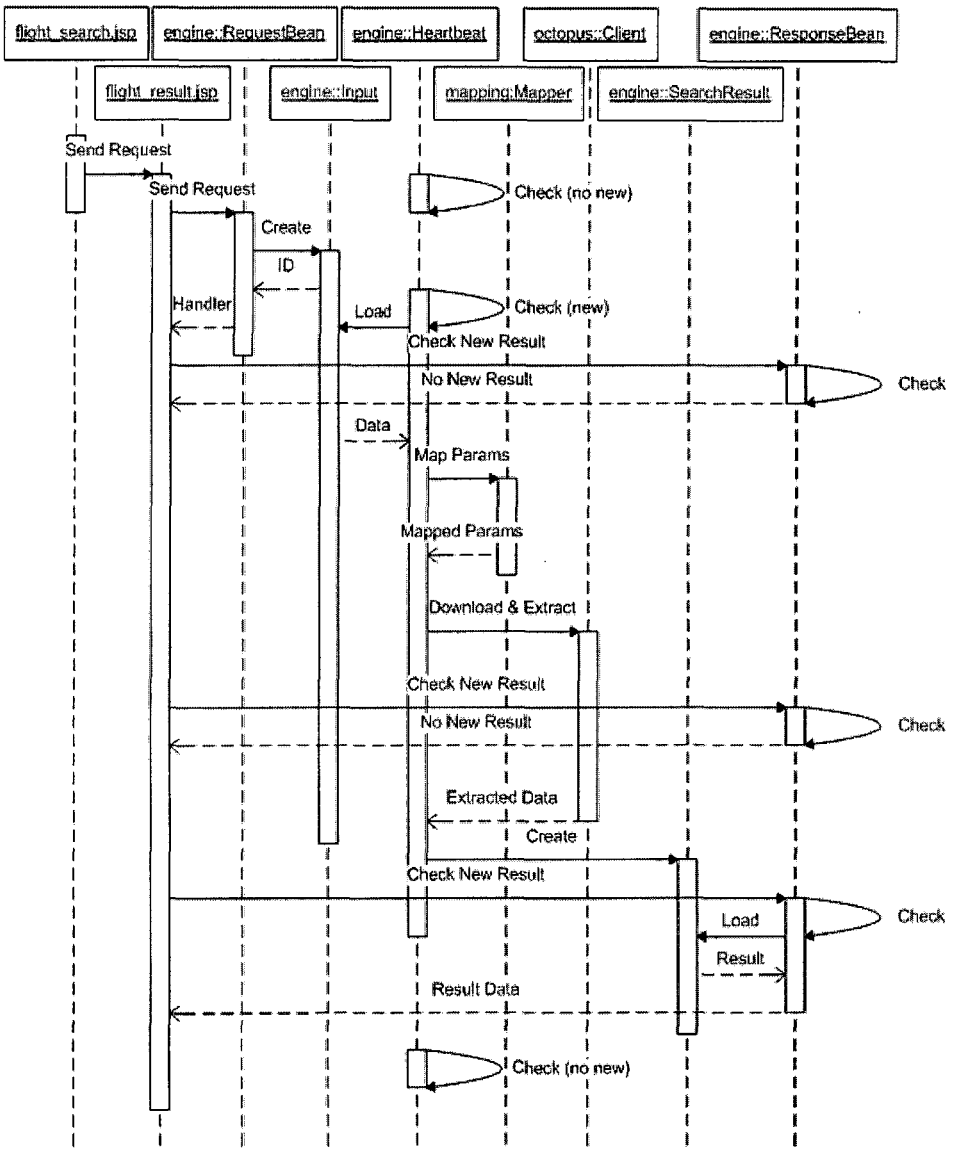


Figure 7.13: Flight Search sequence diagram.

7. If there is a new unprocessed *Input* instance, the *Heartbeat* stateless session bean starts the processing of this instance – the global parameters and values are mapped to local ones in the mapping framework presented by the *Mapper* class.

8. The mapper framework allows a mapping to execute a JavaScript code. It can be executed only out of the application server box. The execution of a JavaScript code provides the Octopus server that is accessible via the *OctopusClient* class. If there is any JavaScript code, a method of the *OctopusClient* class is called to execute the code.

9. The *OctopusClient* class connects with the Octopus server via the JBoss Remoting library and sends the JavaScript code to execute.

10. The Octopus server receives the code to execute and sends it to the Rhino library that executes the JavaScript code.

    The results is sent back to the Octopus server that returns it to the *OctopusClient* class, which then returns the result to the mapping framework.

11. The started processing continues with the download and extraction of data from local search engines. The download and extraction process has to be executed out of the application server box. The download and extraction process of the Octopus server is accessible via the *OctopusClient* class. The *OctopusClient* class is called for each local search form.

12. The *OctopusClient* class connects with the Octopus server via the JBoss Remoting library and sends the corresponding navigation, information on how to download data, and extraction, information how to extract data, with parameters.

13. The Octopus server sends the received navigation and extraction to the Lixto VW & Navigator tool.

    The downloaded and extracted data are sent back to the Octopus server that returns the results to the *OctopusClient* class which returns it to the *Heartbeat* stateless session bean.

14. The received data, that are received asynchronously, are saved to the *Result* entity bean, that is persisted to the database.

15. Meanwhile the result page periodically asks the *ResultBean* stateless session bean for new results with the handler.

16. The *ResultBean* stateless session bean returns a result if there are any unsent data with the specified handler, i.e. the ID.

    If there are no data, a message "*no data*" is sent back to the *ResultBean* stateless session bean that returns it to the result page.

    If there are any unsent data, they are sent back.

    If all processes for the specified handler are finished and there are no unset data, a '*no more results*' message is sent.

The described data flow allows to process requests and to provide results asynchronously – if more requests are received, there are more *Input* entity beans persisted, i.e. there are more result page instances with different handlers. For each handler there are persisted *Result* entity beans.

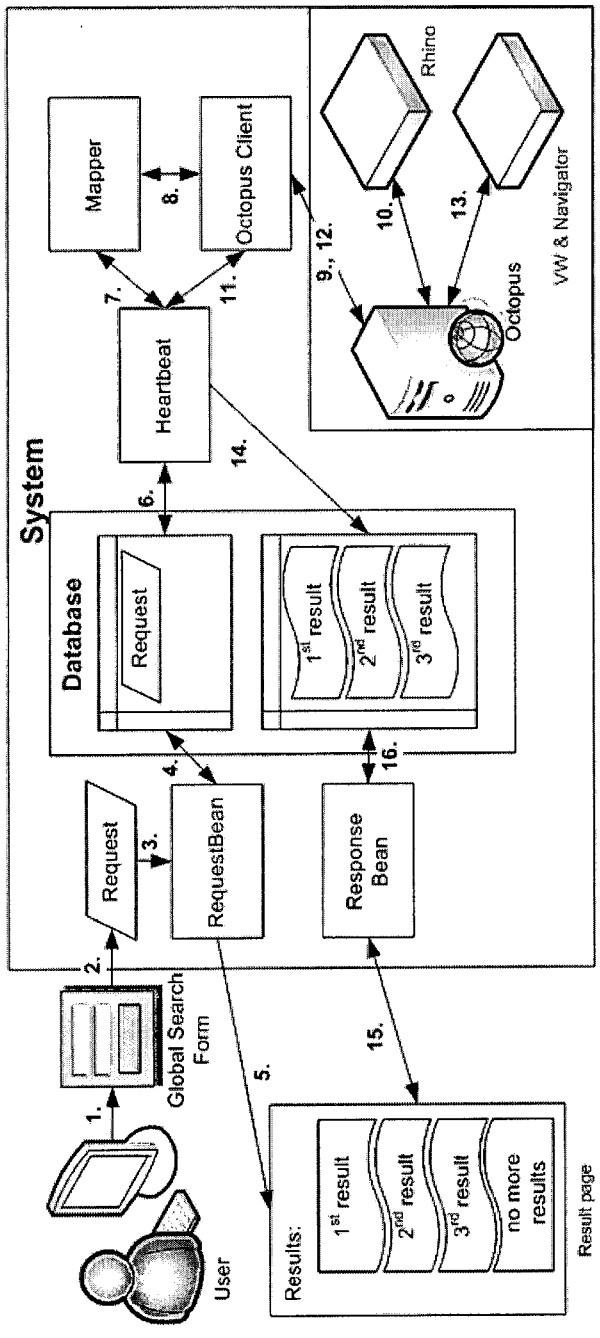


Figure 7.14: Data flow diagram.

Chapter 8

Flight Metasearch Case Study

8.1 Introduction

Flight metasearch solution makes the searching of flights on the web very easy and transparent. A user does not have to check tens of flight booking pages and can choose the most suitable flight for him/her self on a result page that contains flight data from several flight booking systems.

On-line flight booking is an area with well defined parameters and values, results are well structured. Also in terms of coverage, as almost all airlines have their own on-line booking system, this area is suitable for a metasearch solution, i.e. a solution that provides a global search form, where an input is entered only once, this is mapped to particular flight booking systems or flight search engines and results are gathered and transformed into a unified overview form that allows sorting and comparing the received flights data.

Every on-line booking system essentially contains the same parameters like point and time of departure and arrival and number of passengers. This situation is an advantage for a metasearch solution because it simplifies the process of creating a global search form, a mapping between the global search form and a particular booking system, and the final merging of result data.

A global search form contains the same parameters as all particular booking systems. Values of a global form parameter are merged values of the parameter on particular booking systems.

A part of the result is also a *deep link*, i.e. a link that points to the corresponding original page and allows the completion of a booking process for the last step from the search, select and buy steps.

8.2 Requirements

Requirements for the flight search case study are straightforward – to create a metasearch solution that is able to aggregate results from two airline booking

systems.

The flight metasearch prototype covers the following steps:

1. The prototype sends requests and processes results from SkyEurope and Ryanair booking systems.
2. Creating a global search form interface and mappings to local search forms that allow to enter all data that are available on the search forms of the handled booking systems.
3. To merge results from both booking systems and present the result in HTML.

8.3 Form Descriptors

The first step is to create form descriptors for each processed booking system and the global search form.

All forms contain the same parameters: origin and destination airport, departure and return date, and number of passengers.

8.3.1 SkyEurope

The SkyEurope [Sky06a] search form is shown in Figure 8.5; it is analysed in Section 2.5.

The appropriate form descriptor is shown (only its important parts) in Figure 8.1 and contains the following parts:

- a list of origin airports on line 3
- a day of the departure date as a list on line 7
- a month and a year of the departure date as a list on line 11
- a number of adults as a list on line 15
- a number of infants as a list on line 19

The parts for the origin parameters are similar to the departure parts.

8.3.2 Ryanair

The Ryanair [Rya06] search form is shown in Figure 8.5; it is analysed in Section 2.5.

The appropriate form descriptor is shown (only its important parts) in Figure 8.2 and contains the following parts:

- a list of departure airports on line 3
- a day of the departure date as a list on line 7
- a month and a year of the departure date as a list on line 11
- a number of adults as a list on line 15

```
<description name="SkyEurope">
...
        <param list="on" name="_ctl1:gtiCboDest">
                <value>AMS</value>
                <value>ATH</value>
...
        <param list="on" name="_ctl1:cboDepDay">
                <value>01</value>
                <value>02</value>
...
        <param list="on" name="_ctl1:cboDepMonthYear">
                <value>2006/3</value>
                <value>2006/4</value>
...
        <param list="on" name="_ctl1:cboAdults">
                <value>1</value>
                <value>2</value>
...
        <param list="on" name="_ctl1:cboInfants">
                <value>0</value>
                <value>1</value>
...
                <value>9</value>
        </param>
</description>
```

Figure 8.1: Important parts of the form descriptor for the SkyEurope search page.

- a number of children as a list on line 19
- a number of infants as a list on line 23

The parts for the origin parameters are similar to the departure parts.

```
1  <description name="Ryanair">
2  ...
3          <param list="on" name="sector1_d">
4                  <value>AAR</value>
5                  <value>ABZ</value>
6  ...
7          <param list="on" name="sector_1_d">
8                  <value>01</value>
9                  <value>02</value>
10 ...
11         <param list="on" name="sector_1_m">
12                 <value>032006</value>
13                 <value>042006</value>
14 ...
15         <param list="on" name="ADULT">
16                 <value>0</value>
17                 <value>1</value>
18 ...
19         <param list="on" name="CHILD">
20                 <value>0</value>
21                 <value>1</value>
22 ...
23         <param list="on" name="INFANT">
24                 <value>0</value>
25                 <value>1</value>
26 ...
27                 <value>9</value>
28         </param>
29 </description>
```

Figure 8.2: Important parts of the form descriptor for the Ryanair search page.

8.3.3 Global Search From

The global search form is shown in Figure 8.5 and it is described in Section 8.5.

The appropriate form descriptor is shown (only its important parts) in Figure 8.3 and contains the following parts:

- a list of departure airports on line 3
- a day of the departure date as a list on line 7
- a month and a year of the departure date as a list on line 11
- a number of adults as a list on line 15

- a number of children as a list on line 19
- a number of infants as a list on line 23

The parts for the origin parameters are similar to the departure parts.

```
1  <description name="GlobalForm">
2  ...
3     <param list="on" name="departure">
4        <value>AAR</value>
5        <value>ABZ</value>
6  ...
7     <param list="on" name="depd">
8        <value>00</value>
9        <value>01</value>
10 ...
11    <param list="on" name="depmy">
12       <value>032006</value>
13       <value>042006</value>
14 ...
15    <param list="on" name="adult">
16       <value>1</value>
17       <value>2</value>
18 ...
19    <param list="on" name="child">
20       <value>0</value>
21       <value>1</value>
22 ...
23    <param list="on" name="infant">
24       <value>0</value>
25       <value>1</value>
26 ...
27 </description>
```

Figure 8.3: Important parts of the form descriptor for the global search page.

8.4 Mappings

A mapping is shown in Figure 8.4 (only important parts).

Lines 6 to 15 define a mapping for the *adult* parameter. The value mapping uses the *sumAdultChildren* script to compute the number of adult passengers for the SkyEurope search form.

8.5 Global Interface

The global interface of the flight metasearch is shown in Figure 8.5. As in most cases, it does not introduce any new element; it is inspired by the existing interfaces of processed search forms of the processed flight booking systems.

From the design point of view, the global interface is on the client side and it sends a request to the server side.

```
<mapping id="FlightsMapping">
...
        <global>
        </global>
...
        <global name="adult">
                <complete name="_ctl1:cboAdults">
                        <local id="SkyEurope">
                </complete>
                <value name="">
                        <script name="sumAdultChildren">
                                <local id="SkyEurope"/>
                        </script>
                </value>
        </global>
...
        <scripts>
                <script name="sumAdultChildren">
                        getGlobalValue('adult') +
                                getGlobalValue('children')
                </script>
        </scripts>
...
</mapping>
```

Figure 8.4: Important parts of a mapping for the flight metasearch.

8.6 Server Connection

The *flight_search.jsp* global search form page and the *flight_result.jsp* result page are JSP pages. They connect to the human interface gateway provided by the engine prototype implementation.

The search form page sends user parameters and values via the POST method to the result page. The result page sends user parameters and values to the search session bean, which returns a handler. The result page then periodically asks for new results, i.e. it calls the result session bean with the handler via the AJAX technology. Received data is dynamically inserted to the page and the status line is updated.

The result page stops asking for new result when the result session bean sends a terminate notification.

8.7 Results

The results are processed in two steps – firstly, results are extracted and secondly, they are presented to an end customer.

8.7.1 Extraction

A parameterisable navigation ends on a result page, so the extraction functionality of Lixto Visual Developer can be used to extract result data as the

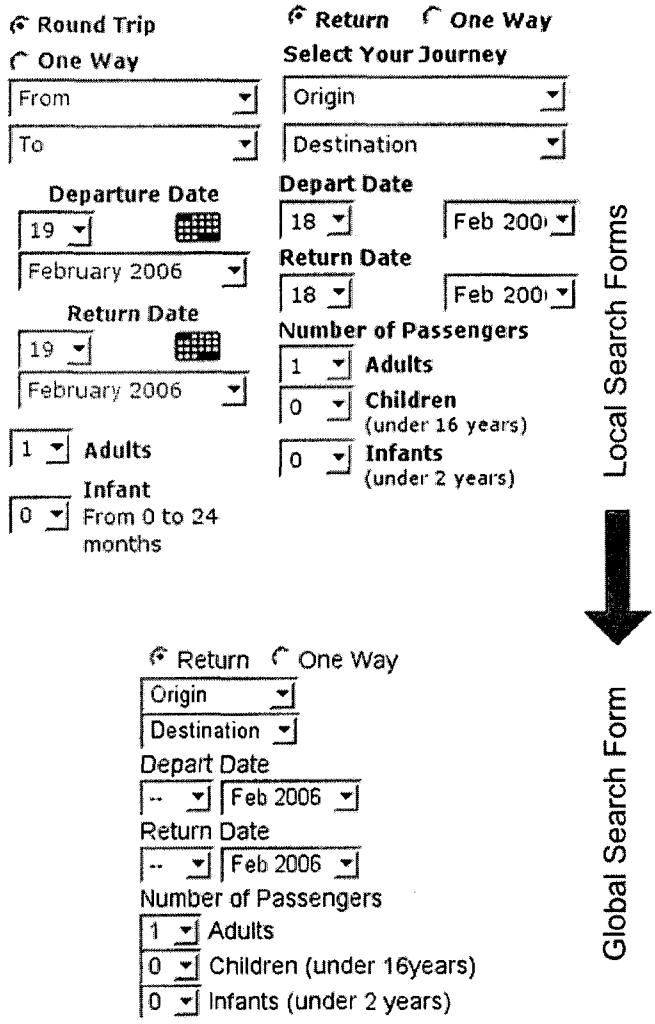


Figure 8.5: The SkyEurope [Sky06a] local search forms (in the upper left corner), the Ryanair [Rya06] local search form (in the upper right corner) and the global search form (at the botom) created by the application designer.

downloading functionality was used to navigate and download a result page; for more details see Section 6.5.3.

An extraction process consists of two parts: an output model and an extraction descriptor.

An output model is generally a schema that defines how an output can look. It is common for all processed airline booking systems. The output example skeleton for the flight metasearch case study is shown in Figure 8.6. Basically the meaning of all items is clear from the concept, the last item *deep-link* on line 9 contains the corresponding deep link, i.e. a URL to the last possible step of the booking process of an airline.

```
1  <flight>
2          <id></id>
3          <airline></airline>
4          <origin></origin>
5          <destination></destination>
6          <departure></departure>
7          <arrival></arrival>
8          <price></price>
9          <deep-link></deep-link>
10 </flight>
```

Figure 8.6: The output example skeleton of the flight metasearch.

An extractor descriptor is created for each airline booking system. It describes how a process of an extraction should be performed, i.e. it is a mapping by rules that identify areas on a HTML page and their content is transfered to the specified output model. For more details see the papers [GKB+04], [BFG01b], [BFG01a] and see Section 4.2.

From the design point of view, the downloading and extraction functionality is performed on the server side.

8.7.2 Presentation

The result data are presented as a HTML page that contains a merged list of all suitable flights found.

The result page has several stages that reflect the behaviour of the engine that fulfil the accessing intermediate results requirement:

- At the beginning, the result page waits for the first result. For example see Figure 8.7.
- The result page receives the first result and displays it. For example see Figure 8.8.
- The result page received two additional results and displays them. For example see Figure 8.9.

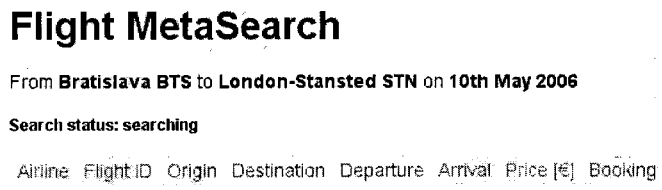


Figure 8.7: The Flight Search result page – the search process started.

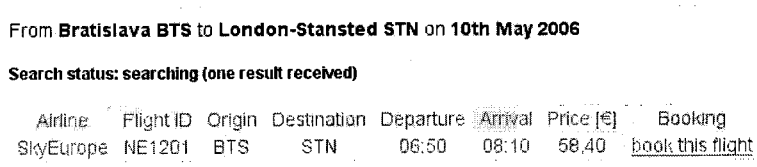


Figure 8.8: The Flight Search result page with one result from SkyEurope.

- The result page received a notification that there are no more results and notifies the end user about it in the status line. For example see Figure 8.10.

The result page uses the AJAX technology, so it is able to dynamically ask for new results and add them without any reloading of the page.

From the design point of view, the presentation of results is done on the client side.

8.8 Conclusion

The flight metasearch case study showed a metasearch solution that satisfies the general requirements mentioned in Section 6.1 and specific requirements from Section 8.2.

The fulfilment of the steps defined in Section 8.2:

1. The Flight Search prototype implementation sends requests and processes results from SkyEurope and Ryanair booking systems – see Section 8.3.1 and Section 8.3.2.

2. To create a global search form interface that allows to enter all data that are available on search forms of the booking systems handled – see Section 8.5.

Flight MetaSearch

From ***Bratislava BTS*** *to* ***London-Stansted STN*** *on* ***10th May 2006***

Search status: searching (3 results received)

Airline	Flight ID	Origin	Destination	Departure	Arrival	Price [€]	Booking
SkyEurope	NE1201	BTS	STN	06:50	08:10	58,40	book this flight
Ryanair	FR2315	BTS	STN	10:00	11:15	29,30	book this flight
Ryanair	FR2317	BTS	STN	21:00	22:00	20,80	book this flight

Figure 8.9: The Flight Search result page with additional two results from Ryanair.

Flight MetaSearch

From ***Bratislava BTS*** *to* ***London-Stansted STN*** *on* ***10th May 2006***

Search status: finished

Airline	Flight ID	Origin	Destination	Departure	Arrival	Price [€]	Booking
SkyEurope	NE1201	BTS	STN	06:50	08:10	58,40	book this flight
Ryanair	FR2315	BTS	STN	10:00	11:15	29,30	book this flight
Ryanair	FR2317	BTS	STN	21:00	22:00	20,80	book this flight

Figure 8.10: The Flight Search result page – the search process finished.

3. To merge results from both booking systems – see Section 8.7.2.

The flight metasearch case study showed how a special-purpose metasearch solution helped an end customer to receive better results in an easy way. The global interface does not significantly differ from the source forms in this case – an end customer can us it immediately.

Chapter 9

Conclusion

This thesis provides an explanation of the main technologies that form the basis of the metasearch world; the existing solutions that defined this field in the past and at present including their advantages, disadvantages, orientation and efficiency, while more thoroughly focusing on the *Lixto MetaSearch* product as the starting point for improvements and new requirements; further it describes the theory of mapping search forms featuring semantic mapping, mapping list support, and date and currency conversion.

It designs a mapping and an execution framework – the main parts of the Snorri metasearch solution. The mapping framework design reflects the described mapping theory and the execution framework design provides a multi-tier, scalable and fail-over solution.

The thesis provides a prototype implementation for the two frameworks. The proposed mapping framework implementation facilitates a simplified development and deployment process of scenarios. The proposed execution framework implementation allows implemented scenarios to be processed in a robust and distributed environment. The prototype implementation represents a solid proof of concept as a base for commercial use.

It contains a case study using the prototype implementation that searches two airline booking systems.

The solution described in this thesis combines the worlds of specialised search engines and metasearch engines. It provides integration of results from back-end databases that are available through particular specialised search engines. Integration includes features such as currency conversion, sorting of results or combining of results from several sources.

The main goal of the solution presented in the thesis is to allow a user to be up-to-date and to choose the most profitable option at any time.

9.1 Requirements Fulfilment

The fulfilment of the requirements from Section 4.5.9 and Section 6.1:

- Faster development of MetaSearch elements. The main aspects of the graphical user interface are mentioned in Section 6.3.6. The mapping framework

significantly simplifies the development and deployment process; it is designed in Chapter 5 and an implementation is described in Section 7.3.3 and Section 7.3.5.

- Monitoring of the whole process. The design is described in Section 6.3.5 and Section 6.4.5, a simplified implementation is shown in Section 7.5.
- Scalability and load balancing. The design is described in Section 6.5.3. The prototype implementation of the engine part shown in Section 7.3.1 uses technologies that provide an industrial standard for creating scalable and load balanced solutions, although the prototype implementation was not tested for it.
- Caching. The caching functionality is designed in Section 6.1.4 and Section 6.5.2. The prototype implementation mentioned in Section 7.3.1 and used technologies provides a base for any caching functionality.
- Source Pruning and Intelligent Pre-loading. The solution is designed in Section 6.3.4 and Section 6.1.4.
- Accessible Intermediate Results. The design is described in Section 6.3.4 and Section 6.5.2. A prototype implementation is shown in Section 8.7.2.
- Session ID Assignment. A session identification assignment is done automatically by using the Lixto navigator Tool for navigations. A solution is explained in Section 5.7 and Section 5.9.
- Multi-page search forms. Multi-page or the wizard style of search forms is supported by using the Lixto navigator Tool for navigations.

9.2 Further Development

Areas for further development include:

- Advanced optimisation and checking techniques. Statistical processing of received data allows the execution framework to perform better optimisation of processing and to check input data based on limits that are adjustable.An expert system can provide sophisticated checking of input data or make expert decisions during the processing phase.
- Grid computing. A huge amount of processed data and requests can be problematic in a multi-tier environment when adding new servers to a cluster does not have the desired effect. In such a case, the grid computing may represent an appropriate solution.
- GUI. A graphical user interface can use features from the mapping theory and design. A combination of features provides the support for an automatic creation of a mapping. It can be used in the preparation of an initial mapping or it can be accessible through a wizard that may focus on a particular aspect of the mapping theory, e.g. the creation of a currency mapping.

9.3 Possible Application and Economic Impact

The proposed solution brings integration of real-time results in all aspects such as different currencies and measure units. The integrated results are classified by different aspects as preferred by an end customer such as price, age and size.

This extends the possibilities of the end customer to search for more sources in a shorter time and it creates a full market picture through the use of one search form.

The solution provides a greater opportunity to find a better value available, thus creating added value for end users.

Naturally, there are many possible application areas, just to name a few: flight booking, car rental, accommodation booking, and shopping.

Acknowledgments

I would like to thank Gábor Farkas, Gerald Ledermüller, Peter Szinek, Tymon Wiedemair, and Viktor Zigo for their hours spent by discussions with me and their priceless comments.

Last but not least, I thank Robert Baumgartner, Georg Gottlob, and Marcus Herzog for their support and helpful advice.

Appendix A

Screenshots

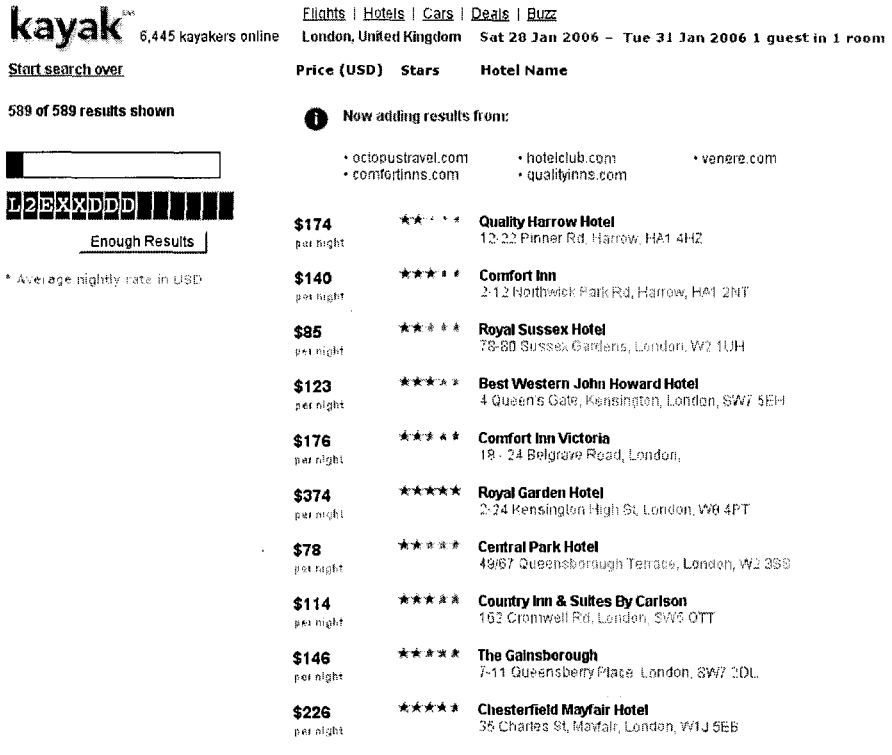


Figure A.1: A kayak.com page with displayed intermediate results, the searching process just started.

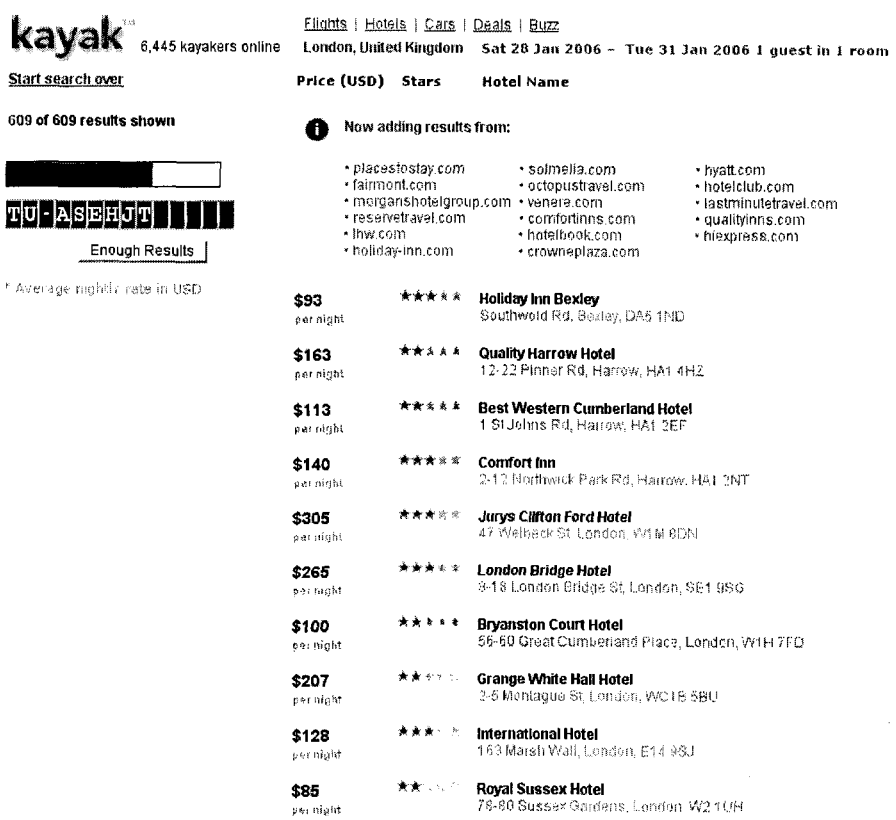


Figure A.2: A kayak.com page with displayed intermediate results, the searching process is finishing.

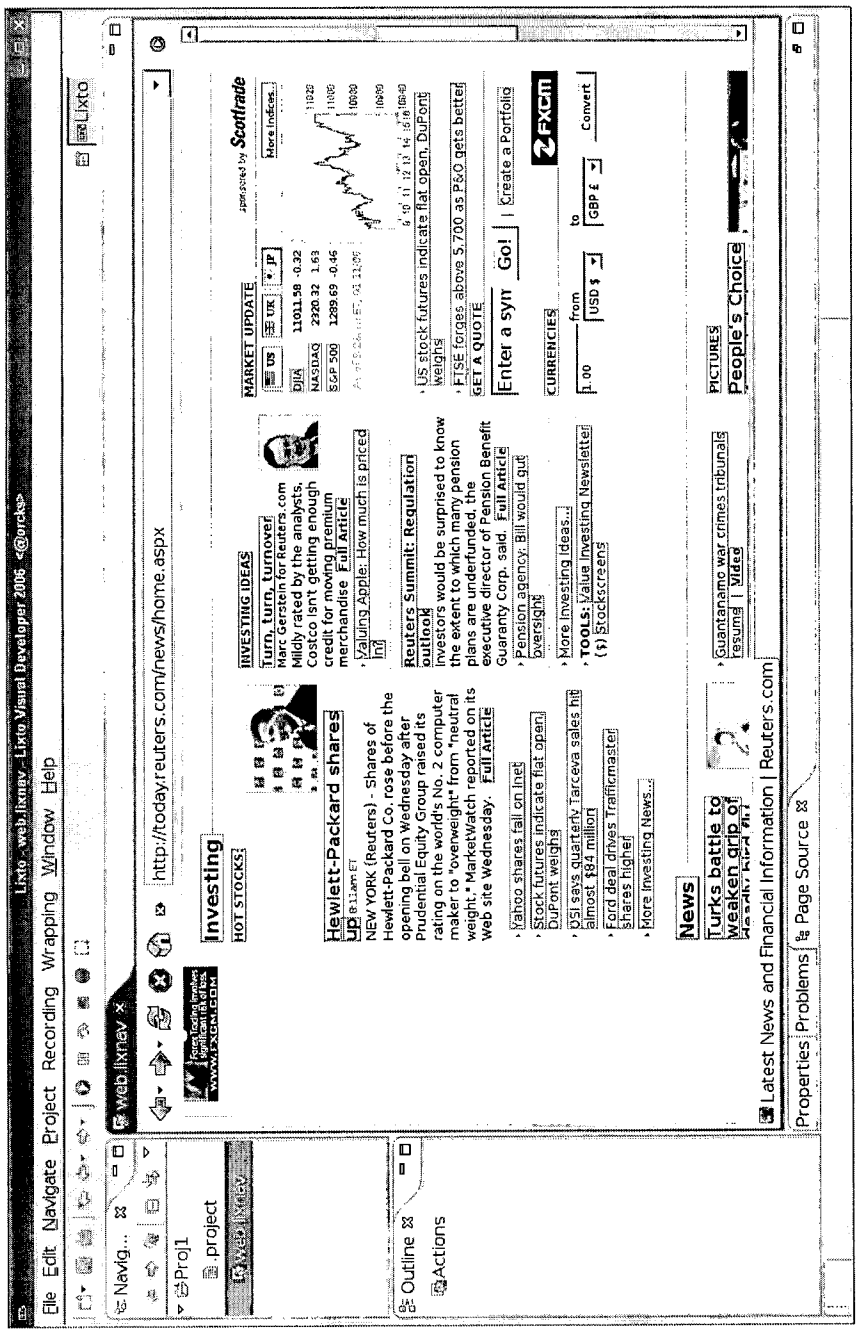


Figure A.3: A screen of the Visual Designer wrapping mode with highlighted anchors.

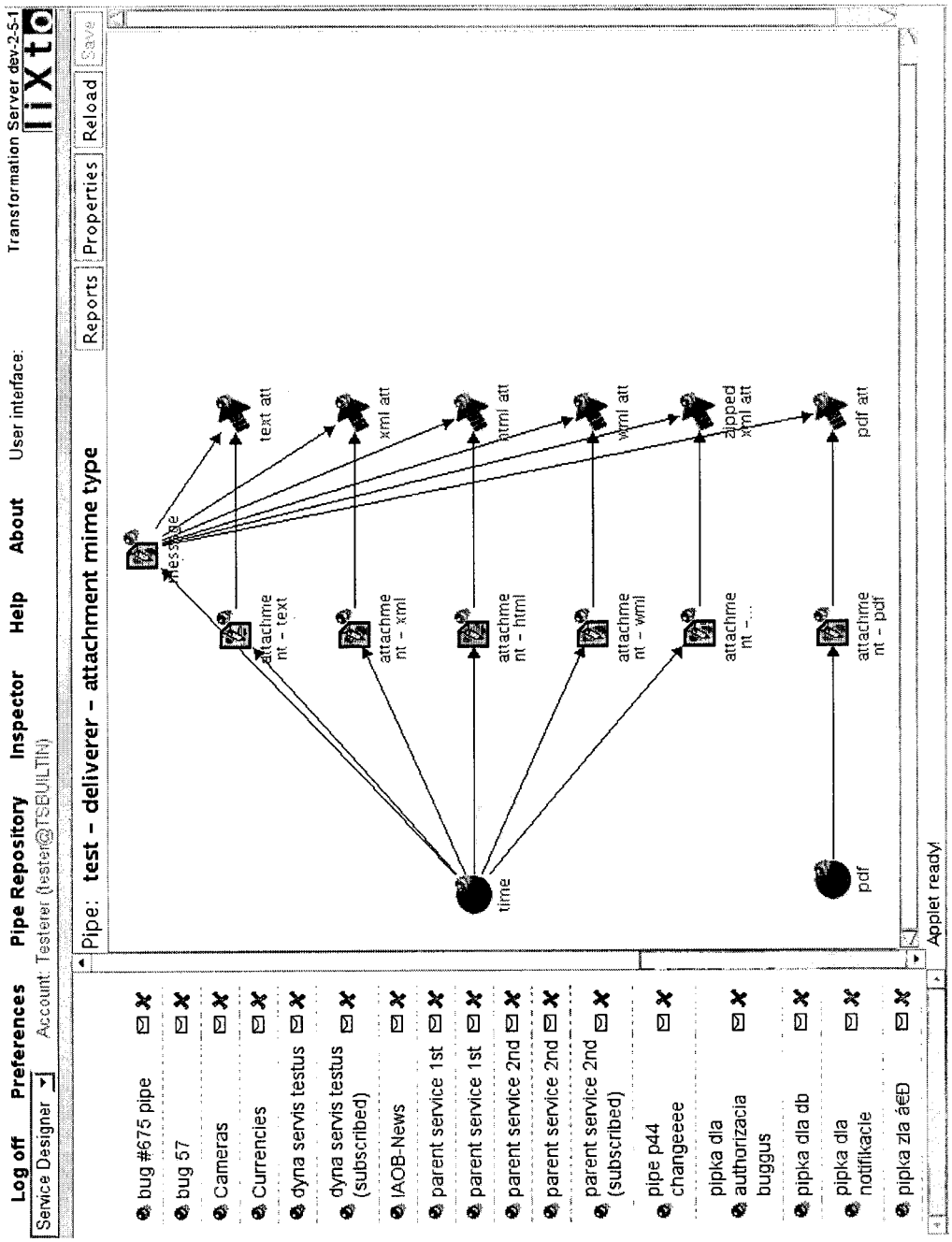


Figure A.4: A screen of the developer interface in Transformation Server. At the menu on the left side are listed developer's pipes. The main area in the middle contains a graphical view of the selected pipe.

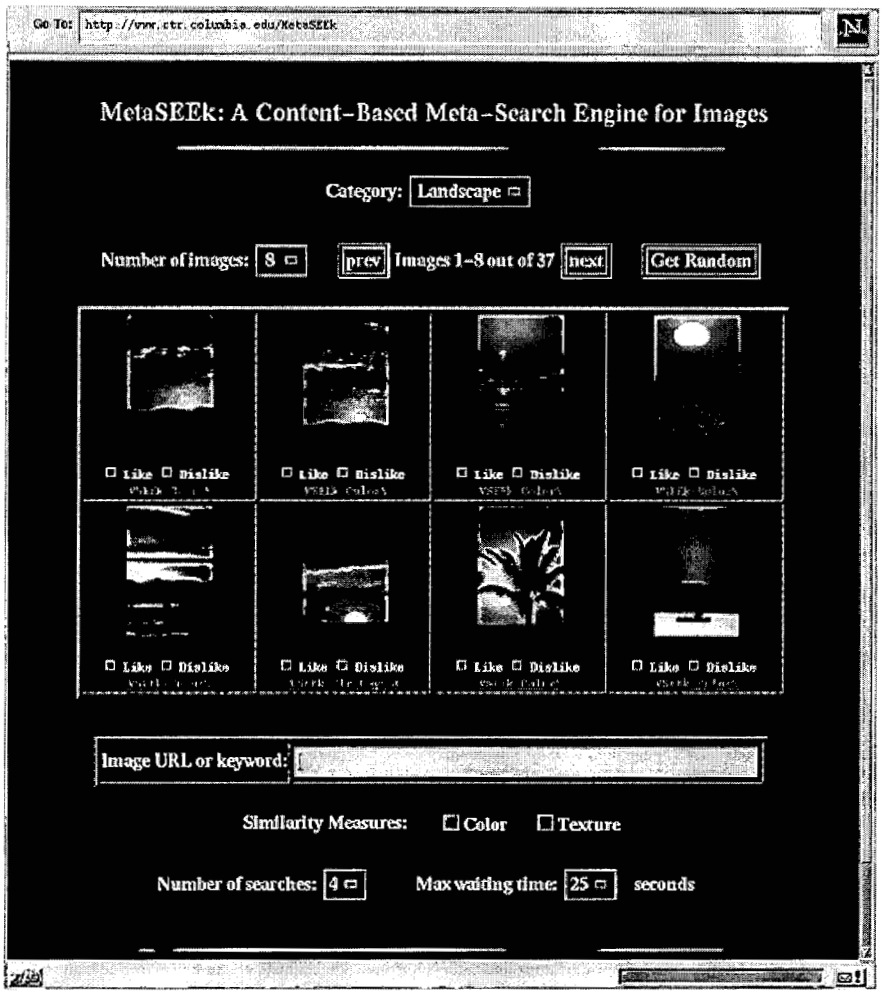


Figure A.5: The MetaSEEk search form page. The search form is based relevance feedback – every picture has *Like* and *Dislike* options. `http://www.ee.columbia.edu/~ana/metaseek/figures/fig1.jpg`

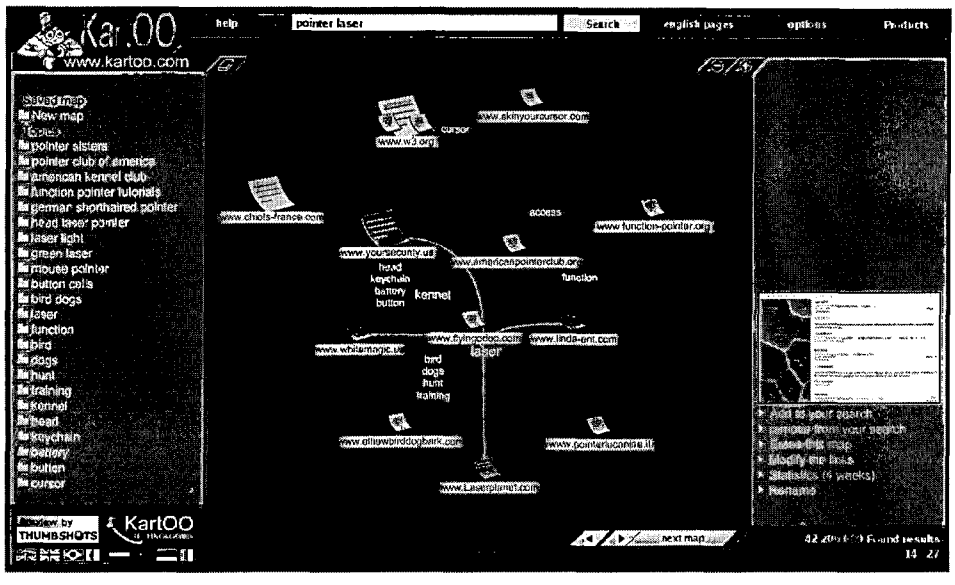


Figure A.6: The graphical results presentation of the KartOO metasearch engine. The nodes represent found pages.

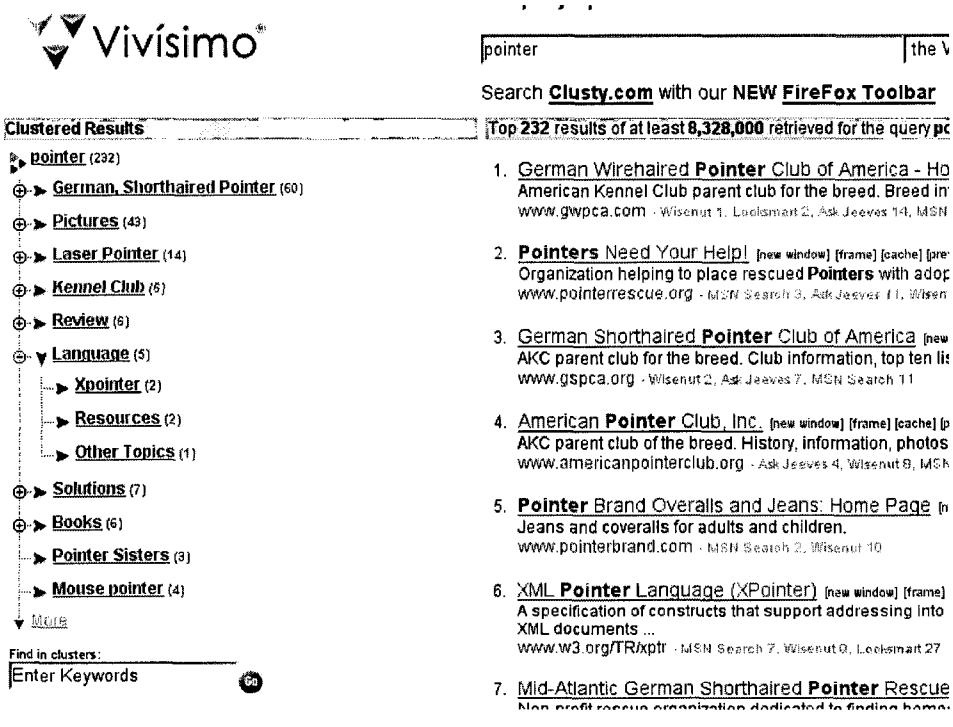


Figure A.7: The clustered results tree of the Vivísimo metasearch engine.

Appendix B

Source Code

```
@Entity
@Table(name="input")
public class Input implements Serializable {
...
  private int id;
  private Timestamp received, accepted, finished;
  private List<ParameterValue> parameters;
  private boolean inProcess, ended;
...
  @Id @GeneratedValue public int getId() {
    return id;
  }
...
  @OneToMany public List<ParameterValue> getParameters() {
    return parameters;
  }
```

Figure B.1: The interesting parts of the Input entity bean source code.

```
@Entity
@Table(name="parameter")
public class ParameterValue implements Serializable {
...
  @Id @GeneratedValue public int getId() {
    return id;
  }
```

Figure B.2: The interesting parts of the ParameterValue entity bean source code.

```
@Entity
@Table(name="result")
public class SearchResult implements Serializable {
...
  private int id;
  private Timestamp created;
  private String data;
  private Input input;
  private boolean sent;
...
  @Id
  @GeneratedValue
  public int getId() {
    return id;
  }
...
  @ManyToOne
  public Input getInput() {
    return input;
  }
```

Figure B.3: The interesting parts of the SearchResult entity bean source code.

```
public @Stateless class RequestBean implements Request {
  @PersistenceContext private EntityManager em;

  public int accept(Map<String, String[]> params) {
    long now = new Date().getTime();

    List<ParameterValue> reqParams =
      new LinkedList<ParameterValue>();
    for (Map.Entry<String, String[]> entry :
      params.entrySet()) {
      if (entry.getValue().length > 0) {
        ParameterValue pv = new ParameterValue(
          entry.getKey(), entry.getValue()[0]);
        reqParams.add(pv);
        em.persist(pv);
      }
    }
  Input input = new Input(reqParams, new Timestamp(now));
  em.persist(input);

  return input.getId();
}
```

Figure B.4: The interesting parts of the Request session bean source code.

```
public @Stateless class HeartbeatBean implements Heartbeat {
  private @Resource SessionContext context;
  private @PersistenceContext EntityManager em;

  public void startTimer() {
    //wait initial 5 sec, then every second
    context.getTimerService().createTimer(5000, 1000, null);
  }
...
  @Timeout public void timeoutHandler(Timer timer)
  {
    List inputs =
      em.createQuery("from Input i where i.inProcess = false").
      getResultList();

    if (inputs.isEmpty()) return;

    Input input = (Input) inputs.get(0);
    input.setInProcess(true);
    input.setAccepted(new Timestamp(new Date().getTime()));

    Map<String, String> reqParams =
      new HashMap<String, String>();
    for (ParameterValue pv : input.getParameters()) {
      reqParams.put(pv.getName(), pv.getValue());
    }

    Map<String, Map<String, String>> locals =
      new Mapper().process(reqParams);

    for (Map.Entry<String, Map<String, String>> entry :
      locals.entrySet()) {
      String source = entry.getKey();
      Map<String, String> sourceParams = entry.getValue();
      String result = new NavigatorClient().
        makeInvocation(source, sourceParams);

      SearchResult sr = new SearchResult(
        new Timestamp(new Date().getTime()), result, input);
      em.persist(sr);
    }
    input.setEnded(true);
    input.setFinished(new Timestamp(new Date().getTime()));
  }
}
```

Figure B.5: The interesting parts of the Heartbeat session bean source code.

```
public @Stateless class ResponseBean implements Response {
  @PersistenceContext
  private EntityManager em;

  public String getResult(String handler) {
    try {
      List results =
        em.createQuery("from SearchResult r where " +
        "r.input.id = " + handler + " and r.sent = false").
        getResultList();

      if (results.isEmpty()) return null;
      SearchResult result = (SearchResult) results.get(0);
      result.setSent(true);

      return result.getData();
    } catch (Throwable e) {
      return "<error/>";
    }
  }
}
```

Figure B.6: The interesting parts of the Response session bean source code.

```
<%!
    private Result result = null;
    public void jspInit () {
        InitialContext ctx = new InitialContext();
        result = (Result) ctx.lookup(
            "prototype/ResultBean/remote");
    }
%>
<%
    String handler = request.getParameter("handler");
    String ret = result.getResult(handler);
%>

<data>
    <% if (ret == null) { %>
        <noresult/>
    <%} else {%>
        <result><%=ret%></result>
    <%}%>
</data>
```

Figure B.7: The interesting parts of the Result JSP page source code.

```
1  @WebService
2  @SOAPBinding(style=Style.RPC)
3  public interface Result extends Remote {
4    @WebMethod String getResult(String handler);
5  }
```

Figure B.8: The interesting parts of the Result web service interface source code.

```
1  @Stateless
2  @WebService(endpointInterface=
3    "snorri.app.interfaces.Result")
4  public class ResultBean implements Result
5  {
6    @EJB Response response;
7
8    public String getResult(String handler) {
9      return response.getResult(handler);
10   }
11 }
```

Figure B.9: The interesting parts of the Result web service interface source code.

Bibliography

[Ama06] Amazon. Search form page, 2006. http://www.amazon.com.

[Apa05a] Apache. Struts key technologies, December 2005. http://struts.apache.org/struts-action/userGuide/preface.html.

[Apa05b] Apache. Tomcat servlet API, 2005. http://tomcat.apache.org/tomcat-5.5-doc/servletapi/index.html.

[Apa06a] Apache. Jmeter project page, 2006. http://jakarta.apache.org/jmeter.

[Apa06b] Apache. Struts page, 2006. http://struts.apache.org/.

[Aut06] Autolocate. Use car search page, 2006. http://parkers.autolocate.co.uk.

[BBC97] M. Beigi, A. Benitez, and S. Chang. Metaseek: A content-based meta search engine for images, 1997.

[BBC98] A. Benitez, M. Beigi, and S. Chang. Using relevance feedback in content-based image metasearch, 1998.

[BEA03] BEA. WebLogic workshop help, 2003. http://e-docs.bea.com/workshop/docs81/doc/en/core/index.html.

[BFG01a] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Declarative information extraction, Web crawling, and recursive wrapping with Lixto. *Lecture Notes in Computer Science*, 2173, 2001.

[BFG01b] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with Lixto. In *The VLDB Journal*, pages 119–128, 2001.

[Car06] Carland. Search page, 2006. http://www.carland.com.

[CCH03] Jared Cope, Nick Craswell, and David Hawking. Automated discovery of search interfaces on the web. In Xiaofang Zhou and Klaus-Dieter Schewe, editors, *The Fourteenth Australasian Database Conference*, volume 17 of *Conferences in Research and Practice in Information Technology*, Adelaide, Australia, 2003. http://research.microsoft.com/users/nickcr/pubs/cope_adc03.pdf.

[Che06a] Checkfelix. FAQ page, 2006. http://www.checkfelix.com/faq.htm.

[Che06b] Checkfelix. Search form page, 2006. http://www.checkfelix.com.

[CHT99] Nick Craswell, David Hawking, and Paul B. Thistlewaite. Merging results from isolated search engines. In *Australasian Database Conference*, pages 189–200, 1999.

[Con99] World Wide Web Consortium. XSL transformations XSLT specification, 1999. http://www.w3.org/TR/xslt.

[Con00] World Wide Web Consortium. XHTML specification, 2000. http://www.w3.org/TR/xhtml1.

[Con04a] World Wide Web Consortium. Extensible markup language XML 1.0 (third edition) specification, 2004. http://www.w3.org/TR/2004/REC-xml-20040204.

[Con04b] World Wide Web Consortium. Web services architecture, 2004. http://www.w3.org/TR/ws-arch.

[Con05a] World Wide Web Consortium. XForms 1.1 working draft, 2005. http://www.w3.org/TR/xforms11.

[Con05b] World Wide Web Consortium. XML path language (XPath) specification, 2005. http://www.w3.org/TR/xpath20.

[Con05c] World Wide Web Consortium. Xquery specification, 2005. http://www.w3.org/TR/xquery.

[Con05d] World Wide Web Consortium. XSL transformations XSLT version 2.0 specification, 2005. http://www.w3.org/TR/xslt20.

[Con06a] World Wide Web Consortium. Soap, 2006. http://www.w3.org/TR/soap12-part1.

[Con06b] World Wide Web Consortium. Web services description language (wsdl), 2006. http://www.w3.org/TR/wsdl20/.

[Con06c] World Wide Web Consortium. XForms page, 2006. http://www.w3.org/MarkUp/Forms/.

[Cou06] Countrbookshop. Advanced search page, 2006. http://www.countrybookshop.co.uk/search/advancedsearch.phtml.

[DH97] Daniel Dreilinger and Adele E. Howe. Experiences with selecting search engines using metasearch. *ACM Transactions on Information Systems*, 15(3):195–222, 1997.

[Fet06] Fetch. Products page, 2006. http://www.fetch.com/solutions.asp?sub=sol-products.

[Gau97] Susan Gauch. Data discovery on the information highway, 1997.

[GKB+04] G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The Lixto data extraction project — back and forth between theory and practice. In *Symposium on Principles of Database Systems 2004*, 2004.

[GKP02] G. Gottlob, Ch. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB 2002)*, pages 95–106, Hong Kong, China, 2002. Morgan Kaufmann.

[GKP03a] G. Gottlob, Ch. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2003)*, San Diego, California, USA, 2003. ACM.

[GKP03b] G. Gottlob, Ch. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency. In *Proc. 19th Int. Conf. on Data Engineering (ICDE 2003)*, Bangalore, India, 2003. IEEE Computer Society.

[GLBG99] Eric J. Glover, Steve Lawrence, William P. Birmingham, and C. Lee Giles. Architecture of a metasearch engine that supports user information needs. In *Eighth International Conference on Information and Knowledge Management (CIKM'99)*, pages 210–216, Kansas City, MO, November 1999. ACM Press.

[GLG+99] Eric J. Glover, Steve Lawrence, Michael D. Gordon, William P. Birmingham, and C. Lee Giles. Web search – your way. *Communications of the ACM*, 1999. accepted for publication.

[Glo01] Eric J. Glover. *Using Extra-topical User Preferences to Improve Web-based Metasearch.* PhD thesis, The University of Michigan, 2001.

[Goo06a] Google. Froogle online shopping page, 2006. `http://froogle.google.com`.

[Goo06b] Google. Help center page, 2006. `http://www.google.com/help/basics.html`.

[Goo06c] Google. Search form page, 2006. `http://www.google.com`.

[HC03] Bin He and Kevin Chen-Chuan Chang. Statistical schema matching across web query interfaces. In *SIGMOD Conference*, pages 217–228, 2003.

[HD97] Adele E. Howe and Daniel Dreilinger. SavvySearch: A metasearch engine that learns which search engines to query. *AI Magazine*, 18(2):19–25, 1997.

[Her02] Marcus Herzog. *A Rapid Application Development Environment for XML-based Data Flow Applications.* PhD thesis, Database and Artificial Intelligence Group, Institute of Information Systems, Vienna University of Technology, Austria, February 2002.

[HMYW04a] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Automatic extraction of web search interfaces for interface schema integration. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 414–415, New York, NY, USA, 2004. ACM Press.

[HMYW04b] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Automatic integration of web search interfaces with wise-integrator. *The VLDB Journal*, 13(3):256–273, 2004.

[HZC05] Bin He, Zhen Zhang, and Kevin Chen-Chuan Chang. Towards building a metaquerier: Extracting and matching web query interfaces. In *ICDE*, pages 1098–1099, 2005.

[Jac98] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley, 3nd edition, December 1998.

[Jav02] JavaWorld. Java scripting languages: Which is right for you?, April 2002. `http://www.javaworld.com/javaworld/jw-04-2002/jw-0405-scripts.html`.

[Jav06a] JavaSource. Open source EJB servers, 2006. `http://java-source.net/open-source/ejb-servers`.

[Jav06b] JavaSource. Open source scripting languages in Java, 2006. `http://java-source.net/open-source/scripting-languages`.

[JBo06] JBoss. Application server page, 2006. `http://www.jboss.com/products/jbossas`.

[Jux06] Jux2. Search form page, 2006. `http://www.jux2.com`.

[Kar06] KartOO. Search form page, 2006. `http://kartoo.com`.

[Kay06a] Kayak. Hotel search API overview, 2006. `http://developer.kayak.com/sysinteg/hotel/overview.html`.

[Kay06b] Kayak. Search form page, 2006. `http://www.kayak.com`.

[Kay06c] Kayak. Technology page, 2006. `http://corp.kayak.com/tech.html`.

[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 133–144. ACM Press, 2002. `http://www.cs.wisc.edu/~raghav/paper-309.pdf`.

[LG98] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.

[LG99] Steve Lawrence and C. Lee Giles. Text and image metasearch on the web. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 829–835. CSREA Press, 1999.

[LMK02] K. Lerman, S. Minton, and C. Knoblock. Wrapper maintenance: A machine learning approach, 2002.

[LMS+05] Yiyao Lu, Weiyi Meng, Liangcai Shu, Clement Yu, and King-Lup Liu. Evaluation of result merging strategies for metasearch engines. In *6th International Conference on Web Information Systems Engineering (WISE05)*, pages 53–66, November 2005. http://www.cs.binghamton.edu/~meng/pub.d/Lu_p211.pdf.

[Mam06a] Mamma. About page, 2006. http://www.mamma.com/info/about.html.

[Mam06b] Mamma. Search form page, 2006. http://www.mamma.com.

[Men06] Weiyi Meng. Projects page, 2006. http://www.cs.binghamton.edu/~meng/metasearch.html.

[Met06] MetaCrawler. Search form page, 2006. http://www.metacrawler.com/.

[Moz00] Mozilla. Gecko FAQ, November 2000. http://www.mozilla.org/newlayout/faq.html.

[Moz05] Mozilla. About mozilla, October 2005. http://www.mozilla.org/about.

[Moz06] Mozilla. Javascript for Java, 2006. http://www.mozilla.org/rhino/.

[MWR+05] Dheerendranath Mundluru, Zonghuan Wu, Vijay Raghavan, Weiyi Meng, and Hongkun Zhao. Automatically extracting subsequent response pages from web search sources. In *IEEE Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources*, November 2005. http://www.cs.binghamton.edu/~meng/pub.d/ICDMWorkshop05.pdf.

[MYL02] Weiyi Meng, Clement T. Yu, and King-Lup Liu. Building efficient and effective metasearch engines. *ACM Computing Surveys*, 34(1):48–89, 2002.

[Nau04] Felix Naumann. Detecting duplicate objects in XML documents. In *Proceedings of the 2004 international workshop on Information quality in informational systems*, pages 10–19. ACM Press, 2004. http://www.informatik.hu-berlin.de/mac/publications/IQIS04.pdf.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[Ope06] Opensymphony. Webwork page, 2006. http://www.opensymphony.com/webwork.

[Ora06] Oracle. Oracle application server EJB 3.0, 2006. http://www.oracle.com/technology/tech/java/ejb30.html.

[Pen06] Penshop. Shop page, 2006. http://www.penshop.co.uk.

[PMHY04] Qian Peng, Weiyi Meng, Hai He, and Clement T. Yu. Wisecluster: clustering e-commerce search engines automatically. In *WIDM*, pages 104–111, 2004.

[Pro06] ProFusion. Search form page, 2006. http://www.profusion.com.

[RAJ02] Ed Roman, Scott W. Ambler, and Tyler Jewell. *Mastering Enterprise JavaBeans.* Willey Computer Publishing, 2nd edition, 2002. http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471417114.html.

[Ros04] Florian Rosenberg. A congurable deep web metasearch engine based on Lixto. Master's thesis, Software Engineering, Fachhochschul-Diplomstudiengang, Hagenberg, Austria, September 2004.

[Rya06] Ryanair. Search form page, 2006. http://www.ryanair.com.

[SE95] E. Selberg and O. Etzioni. Multi-service search and comparison using the MetaCrawler. In *Proceedings of the 4th International World-Wide Web Conference*, Darmstadt, Germany, December 1995.

[Sea06] Search. Search form page, 2006. http://www.search.com.

[Sim06] Kai Simon. Meta search engine systems page, 2006. http://www.informatik.uni-freiburg.de/~ksimon/Meta_Search_Engines.html.

[Sky06a] SkyEurope. Search form page, 2006. http://www.skyeurope.com.

[sky06b] skyscanner. Main page, 2006. http://www.skyscanner.net.

[sPK89] Mikuláš Popper and Jozef Kelemen. *Expertné systémy (Expert Systems, in Slovak).* Alfa, Bratislava, 1989.

[Ste01] Robert Steele. Techniques for specialized search engines. In *Proc. Internet Computing 2001*, Las Vegas, June 25–28 2001.

[Sun01] Sun. Java message service specification 1.0.2, August 2001.

[Sun03a] Sun. Java native interface specification 1.1, 2003. http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/jniTOC.html.

[Sun03b] Sun. JSR 153, enterprise javabeans 2.1 specification, November 2003. http://jcp.org/en/jsr/detail?id=153.

[Sun05a] Sun. The Java language specification, third edition, 2005. http://java.sun.com/docs/books/jls/third_edition/html/intro.html.

[Sun05b] Sun. JSR 220, enterprise javabeans 3.0 specification, December 2005. http://jcp.org/en/jsr/detail?id=220.

[Sun05c] Sun. JSR 244, enterprise edition 5 (Java EE 5) specification, November 2005. http://jcp.org/en/jsr/detail?id=244.

[Sun05d] Sun. White paper iPlanet portal server - session service, 2005. http://developers.sun.com/sw/docs/wpapers/portal/session.html.

[Sun06a] Sun. Java server faces specification 2.1 final draft 2, February 2006.

[Sun06b] Sun. Java server pages specification 2.1 final draft 2, February 2006.

[Sun06c] Sun. Model-View-Controller page, 2006. http://java.sun.com/blueprints/patterns/MVC.html.

[The05] TheServerSide. Application server matrix, 2005. http://www.theserverside.com/articles/article.tss?l=ServerMatrix.

[Viv06a] Vivísimo. Content integrator specifications product page, 2006. http://vivisimo.com/html/vci-specs.

[Viv06b] Vivísimo. Search form page, 2006. http://vivisimo.com.

[Viv06c] Vivísimo. Velocity product page, 2006. http://vivisimo.com/html/velocity.

[VM02] Andrej Vckovski and Michel Mathis. Scripting for Java. In *9th Annual Tcl/Tk Conference*, 2002.

[W3C99a] W3C. Hypertext markup language specification 4.01, December 1999. http://www.w3.org/TR/REC-html40.

[W3C99b] W3C. Hypertext transfer protocol specification 1.1, June 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[Wei04] Qian Peng Weiyi. Clustering e-commerce search engines, 2004.

[Wik06] Wikipedia. Database, 2006. http://en.wikipedia.org/wiki/Database.

[WMYL01] Zonghuan Wu, Weiyi Meng, Clement T. Yu, and Zhuogang Li. Towards a highly-scalable and effective metasearch engine. In *World Wide Web*, pages 386–395, 2001.

[WYDM04] Wensheng Wu, Clement T. Yu, AnHai Doan, and Weiyi Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD Conference*, pages 95–106, 2004.

[Yah06] Yahoo. Search tips page, 2006. http://help.yahoo.com/help/us/ysearch/basics/basics-04.html.

[ZMW+05] Hongkun Zhao, Weiyi Meng, Zonghuan Wu, Vijay Raghavan, and Clement Yu. Fully automatic wrapper generation for search engines, May 2005.

MSc. Ondrej Jaura

CONTACT

Blagoevova 16
851 04 Bratislava
Slovakia

tel.: +421 905 987 874
e-mail: ondrej@realtime.sk
www: http://realtime.sk/~ondrej

PROFILE

alphabetical skills listing:
Ajax, Ant, Bash, C, CSS, CVS, Delphi, DHTML, Eclipse, EJB, Hibernate, HTML, J2EE, Java, JavaBeans, JavaScript, JBoss, JSP, JUnit, LaTeX, Log4J, MVC, Oracle AS&DB, Pascal, PHP, PL/SQL, PostgreSQL, Scrum, Servlet, SQL, Subversion, Tomcat, UML, Velocity, Xalan, Xerces, XHTML, XML, XSL-FO, XSLT
communication skills:
Slovak - native, English, Czech - fluent. German - good
driving licence:
B

EDUCATION

Database and Artificial Intelligence Group,
Institute of Information Systems, Computer Science Department,
Vienna University of Technology, Vienna, Austria **2004 - now**
PhD studies

Faculty of mathematics, physics and informatics
Comenius University, Bratislava, Slovakia **1998 - 2003**
Field: Computer Science
Specialization: Algorithms and paralel computing, Artificial intelligence
Diploma thesis: Emergence of vocabulary in a society of agents
The Master of Science degree attained in June 2003.

Secondary school (Gymnázium) P.O.Hviezdoslava, Kežmarok **1993 - 1997**

EXPERIENCE

Lixto GmbH., Vienna, Austria, *http://www.lixto.com*
software architect and developer **February 2004 - now**
Product Transformation Server. Java, J2EE, XML, projects from Apache Software Foundation and Oracle DB and AS. Design and analysis of a distributed process workflow engine. Implementation, testing and bug fixing. Installation. CVS, bug tracking system. Work in an international team (approx. 15 members) – Project Management with Scrum.

Database and Artificial Intelligence Group,
Institute of Information Systems, Computer Science Department,
Vienna University of Technology, Vienna, Austria, *http://www.dbai.tuwien.ac.at*
developer **January 2001 - January 2004**
Project Lixto, product Transformation Server. Developing in Java, XML, projects from Apache Software Foundation and Oracle AS and DB. System design, implementation, testing and bug fixing. CVS, bug tracking system. Work in an international team (approx. 10 members).

Datalock a.s., Bratislava, Slovakia *http://www.datalock.sk*
developer-analyst **2001 - June 2003**
Design and implementation of the Horec project (Java). Using state-of-the-art technologies. Independent solution seeking. Development of prototypes.

developer **November 1999 - 2001**

Projects Monetka (Borland Delphi) and Horec (Java). Design, implementation, testing, bug fixing, installation, customer support. Teamwork (2 - 5 members).

Borland s.r.o., Bratislava, Slovakia *http://www.borland.sk*

lecturer **November 2001 - March 2002**

Training courses "Introduction to Java" a "Java for advanced". Preparation and presentation of course materials.

PROFESSIONAL EXPERIENCE

- *computer languages:* Java, Pascal, C, LISP
- *developer IDE:* Eclipse, Borland JBuilder 4-X
- *J2EE:* EJB 3.0 and 2.x, XDoclet, Sevlet, JSP
- *J2EE application servers:* JBoss, Oracle AS
- *web:* XHTML, CSS, JavaScript, HTML, PHP, DHTML, Ajax
- *XML:* XML, XSLT, XSL-FO
- *DB:* SQL, PL/SQL
- *DB servers:* Oracle DB, PostgreSQL, MySQL
- *apache.org:* Tomcat, Xerces, Velocity, Log4J, Struts, Xalan, FOP
- *operating system:* Linux, Windows

PERSONAL

photography, movies, music, reading books, in-line skating, squash