Identifying, Structuring and Searching HTML Objects

IVAR EMBREUS

Master’s Degree Project
Stockholm, Sweden 2005

TRITA-NA-E05043
Identifying, Structuring and Searching HTML Objects

IVAR EMBREUS

TRITA-NA-E05043

Master’s Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering,
Royal Institute of Technology year 2005
Supervisor at Nada was Martin Hassel
Examiner was Stefan Arnborg
Identifying, Structuring and Searching HTML Objects

Graduation Thesis

Abstract

The World Wide Web as a public source of information contains enormous amounts of data in formats that often are less appropriate to be processed. The lack of tools for extraction of data out of HTML pages has lead to this project. The main issues of this project are related to the different steps in data extraction, namely identification and structuring of HTML objects and comparison of these structures and string attributes in order to search for related parts of different pages in a site.

Further, this project presents a prototype of a software able to extract information from an arbitrary site to a Content Management System (CMS) developed by Dotify AB.

Identifiering, strukturering och sökning av HTML-objekt

Examensarbete

Sammanfattning

Webben som en publik källa av information innehåller enorma mängder av data i format som ofta inte lämpar sig för databehandling. Behovet av ett verktyg för dataextraktion ur html-sidor har lett till detta projekt. De viktigaste delarna av projektet är relaterade till de olika processerna för dataextraction, nämligen identifiering och strukturering av html-sidor samt jämförelse av dessa strukturer och strängattribut för att hitta relaterade delar av olika sidor i en sajt.

Vidare presenteras en prototyp av ett system kapabelt att extraiera information ur en godtycklig sajt till ett Content Management System - CMS utvecklat av Dotify AB.
# Contents

## 1 Related Work
- 1.1 Wrappers and Wrapper Generation ........................................... 1
  - 1.1.1 HTML Aware Systems .................................................. 2
- 1.2 Natural Language Processing ................................................ 2
  - 1.2.1 CRYSTAL System ......................................................... 2
  - 1.2.2 Taxonomy of HTML Segments ......................................... 2
- 1.3 Modeling Based Tools ......................................................... 3
- 1.4 Ontology Based Conversion .................................................. 3
- 1.5 Conceptual-Modeling .......................................................... 4

## 2 Introduction
- 2.1 Background ............................................................................. 5
- 2.2 Motivations ............................................................................. 6
- 2.3 System Overview .................................................................... 6
  - 2.3.1 Main Features .................................................................. 6
- 2.4 Problem Definition .................................................................. 7
- 2.5 Delimitation ............................................................................ 8
  - 2.5.1 The HTML Pages Code ................................................... 8
  - 2.5.2 Browsing the Objects ....................................................... 8
  - 2.5.3 Nested HTML Objects ..................................................... 9
  - 2.5.4 The Prototype Development .......................................... 10
  - 2.5.5 The Final Result ............................................................. 10

## 3 Methods
- 3.1 Extracting Data from Internet ............................................... 11
- 3.2 Parsing HTML Code ............................................................. 12
  - 3.2.1 Parsing HTML Tables .................................................... 13
- 3.3 Structuring HTML Objects .................................................... 15
- 3.4 The Presentation of the Object Tree ....................................... 15
- 3.5 Detecting HTML Objects ....................................................... 16
  - 3.5.1 Appearance Search ......................................................... 17
  - 3.5.2 Appearance Search - Regular Expressions ....................... 17
  - 3.5.3 Searching Embedded HTML-tags ................................... 18
  - 3.5.4 Minimal Matches ......................................................... 18
  - 3.5.5 Maximal Matches ......................................................... 19
  - 3.5.6 Subtree Comparison ....................................................... 19
- 3.6 Conclusions and Comments .................................................. 22

References ..................................................................................... 24
A Description of the Prototype

A.1 Preliminaries ............................................................... 27
A.2 Main Features ............................................................ 27
A.3 The Application GUI ..................................................... 28
A.4 Using the Prototype ...................................................... 28
  A.4.1 The Start Window
    ... and parsing the pages ............................................ 28
  A.4.2 View of the Parsed Page
    ... and browsing individual nodes ................................ 29
  A.4.3 Searching for Similar Objects
    ... or running the Maximal Match Algorithm ..................... 29
  A.4.4 Structuring Candidate Pages
    ... or parsing candidate pages ..................................... 29
  A.4.5 Finding Similar Objects
    ... or running the tree comparison algorithm .................. 30
A.5 Things to be Improved ................................................ 30
A.6 Known Problems ........................................................ 30

B Regular Expression Rules .............................................. 31
  B.1 Rules for Detection of HTML Objects ............................. 31
    B.1.1 HTML Objects .................................................. 32

C Brainstorm ................................................................. 33
  C.1 Brainstorm ............................................................ 33
Chapter 1

Related Work

During the expansion of the Web different tools have been developed and several theoretical works have addressed the issue of Information Extraction\(^1\) (IE) from Web pages. The techniques used extend from natural language processing to machine learning, passing by information retrieval, database and ontology based techniques. In this chapter a short presentation of the most common approaches will be made.

There are several different approaches and goals in structure analysis of Web data. In this section some interesting works will be referenced. Not all of them form a base for this thesis, but some of the ideas shown in many papers have been used as source of inspiration.

The techniques discussed in this section can be separated into groups according the approaches used. The groups presented here are:

- Wrappers and Wrapper Generation
  - HTML Aware Tools
- Natural Language Processing
- Modeling Based Tools
- Ontology Based Tools
- Conceptual-Modeling

1.1 Wrappers and Wrapper Generation

A Wrapper is a tool that is capable of recognizing data of interest among a number of other things (strings, texts) of the same type fastened or closely grouped together. The data of interest is separated by the Wrapper according to rules that are used to describe the objects like mark-up tags, inline code, navigation hints etc.

Perhaps the biggest shortcoming with Wrappers is the difficulty in writing and maintaining them. In order to overcome this problem many tools for Wrapper generation have been developed. Tools like (Baumgartner et al. 2001, Mecca et al. 1998, Crescenzi et al. 1998) are examples of languages specially designed for wrapper generation. Java based Extraction and Dissemination of Information (JEDI) (Huck et al. 1998) is another tool that besides works for any textual data.

\(^1\)Message Understanding Conferences (MUC) defines Information Extraction as the identification of instances of a particular class of events or relationships in a natural language text, and the extraction of the relevant arguments of the event or relationship.
Chapter 1. Related Work

The use of Wrapper generators is one of the most common approaches in Web data extraction. Minerva (Crescenzi and Mecca 1998) is one of the most known tools for wrapper generation.

The wrapper approach is basically composed of three steps:

1. extracting attributes and values from the text,
2. instantiation of objects according to some rules and
3. structuring the objects in some kind of data structure.

The use of wrappers is a common approach in the area of Information Extraction and Data Integration.

1.1.1 HTML Aware Systems

World Wide Web Wrapper Factory (W4F) (Azavant and Sahuguet 2000) and XWRAP are good examples of tools that are used to generate Wrappers.

XWRAP (Liu et al. 2000) is a interactive tool that is used for wrapper generations having the Web as information source. It makes use of the content property of HTML tags, considering it as meta-data for the informational data content in Web pages and explicitly writes it in XML documents. The XML documents are then used in a process for query-based filtering of content.

1.2 Natural Language Processing

Natural Language Processing (NLP) as an information extraction system performs the task of identifying references to a specific context based on a knowledge source (often called a Dictionary). This dictionary contains the definitions and rules for text analysis containing vocabulary, semantic classes and writing style characteristic to the context in question (Soderland 1997) and works as knowledge sources of a Information Extraction system. The difficulty and time consumption in writing the dictionary entries (rules) by hand motivated the development of the CRYSTAL system.

1.2.1 CRYSTAL System

The CRYSTAL system (Soderland et al. 1995) is a system that automatically learns domain-specific rules for information extraction. Fundamentally CRYSTAL prompts a dictionary of “concept-node definitions”2 that allow the identification of correlated material from a training text (corpus).

It uses a technique from machine learning called covering algorithm. The system uses tagged references to the target concepts to distinguish elements of interest in the target text.

The CRYSTAL system is used in domain specific cases, as in (Soderland 1996), one of the domains used as example is news articles about people moving to top corporate management positions and people moving out of those positions. Information considered important, in this case, are the people, their positions, and corporations.

1.2.2 Taxonomy of HTML Segments

Conventional Natural Language processing techniques demand linguistic (syntactically) correct elements and full-dressed sentences which conflicts with the summarized characteristics usually found in Web pages.

---

2Concept-Node Definition (CN’s) is the representation of case-frames in the University of Massachusetts BADGER sentence analyzer.
(DiPasquo 1998) sustains that adding taxonomy for the HTML embedded structure improves the performance of segment categorization. The approach uses a parsing algorithm responsible for the creation of Struct Trees. These Struct Trees regulate the learning process of extracting relevant information from HTML pages by using visual indications given by lists, tables, hyperlinks, etc.

### 1.3 Modeling Based Tools

This class of tools work on the structures embedded in the Web pages, comparing the morphologies of fragments (HTML objects) of the pages.

*Northwestern Document Structure Extractor* (Adelberg 1998) (NoDoSE) works interactively and detects the structures inside HTML documents and extracts their data. NoDoSE is called a semi-automatic system since many decision steps like e.g., data model, are left to the user. With the aid of a GUI the user can break down the Web data in objects of lower complexity. While decomposing the document the user assigns the interesting parts of a page and the system “learns” how to extract the same kind of information from other pages.

(Laender et al. 2002) and (Ribeiro-Neto et al. 1999) uses basically the same idea, invoking clues from the user to determine relevant portions of a Web page, which later is used as a template for finding the same “kind” of information in other Web pages.

### 1.4 Ontology Based Conversion

The use of ontologies\(^3\) arises from the field of Artificial Intelligence (AI) where attempts where made to find ways to describe the context surroundings and the knowledge base of relationships among objects in the environment. The ontologies are the background knowledge (Gruber 1993) that dictate the agreements, assumptions and models of the world. The ontology is applied to recognize constants in the Web pages and to instantiate objects with them.

One of the most representative works based on ontology is the one within the *Virtual Information Processing Agent Research* (VIPAR) project (Potok et al. 2002). The issue of this work is to solve the hard problem of converting HTML to XML using intelligent agents and ontologies. The domain in focus is the Internet newspapers’ sites and the main question is determining a common description able to make the conversion of HTML-data to XML, that works for dissimilar pages. The ontology or description in the VIPAR project includes five basic elements which are listed here as they exemplify the role of the ontologies:

- article meta-data
- traversing directives
- traversal maps
- article delimiters
- article structuring rules

Each item listed is actually a rule or a set of rules telling the action to be taken every time the process is triggered.

\(^3\)The term is borrowed from philosophy. From [http://www.dictionary.com](http://www.dictionary.com)

i) That department of the science of metaphysics which investigates and explains the nature and essential properties and relations of all beings, as such, or the principles and causes of being.

ii) The branch of metaphysics that deals with the nature of being.
The steps executed by the VIPAR system include the usage of agents that retrieve information from Internet sites. These agents perform two basic tasks. The first is to parse HTML code. The second is to perform the rules embedded in the ontologies. Together, these steps allow the agents to automatically traverse a site, search and retrieve relevant pieces of data, convert them to XML and then send the information back to the VIPAR system.

1.5 Conceptual-Modeling

The Conceptual-Modeling (Embley et al. 1998) listed in the beginning of this chapter is actually not a technique of its own. It is predominantly an employment of two other approaches. What makes the concept interesting is the way it uses ontology and wrapper-generation techniques to achieve the goal of extracting data out of Web pages. The application domain in this paper is rather narrow which simplifies the understanding of the study.

In (Embley et al. 1998) the goal is to make the information queryable transforming information in HTML-pages to database entries. This report presents a different and new approach which uses both a wrapper and an ontological specification. The application ontology (see section 1.4 above) works as a conceptual model where the descriptions of interest data, relationships, lexical appearance and context keywords from obituaries are the input data to the ontology parser that generates a database scheme and rules for matching constants and keywords. It also uses a record extractor to remove HTML-tags and separate the records. Further recognizers work on the constants and keywords and extract the objects and their relationships. The population of the database is done using heuristics which put the extracted keywords in relation with extracted constraints. Cardinality constraints in the ontology ascertain the construction of the records and the insertion in the database.

The concept of using an ontology is particularly interesting when working with well defined situations. By changing the ontology it is highly possible to use the concept in different contexts.

What makes this approach not directly useful for this project is the fact that the narrowness of the ontology implies specific domains of application, since the ontology requires restriction of the boundaries of the chosen domain. The ontology even involves the process of deciding if a web page is of interest or not. In the case of this project all the pages are of interest.

\footnote{For illustration, the paper uses textual information from obituaries.}
Chapter 2

Introduction

In this chapter the background, motivations and problem definitions for this project will be introduced. The section System Overview gives an overall illustration of how the system fulfills its assignment. The Background and Motivations sections give a short introduction of the IT-company Dotify AB and its need of an importer module for its Content Management System called Dotify WMS. The Problem Definition section shows how the problem could be divided in several minor phases that together render the solution.

Chapter 1 introduces some related researches made in the field of Information Extraction (IE). The techniques presented in Chapter 1 are not exhaustive. The techniques presented are only some of the most common in the field. Later, in chapter 3, a more detailed discussion of the solutions will be given. A deeper presentation of the different steps executed to achieve the solution (section 2.4 in this chapter) will also be presented from section 3.1 to 3.5. Appendix A introduces the prototype system created for testing the strategies presented in Chapter 3. The last part of this thesis (appendix B.1) describes the rules used to extract HTML objects from Web data and some brainstorm related to an eventual implementation of the system proposed.

2.1 Background

As a consequence of the tremendous expansion of the World-Wide Web, the volume of information being held in HTML format is also increasing. These pages do not offer good possibilities to data manipulation in form of restructuring, information fetching etc. As the Web becomes more and more important in business there is an increasing need of people with know-how in web publishing. But rather commonly the people who want material published on the Web do not have the right knowledge to do it. One solution is to use a Content Management System as a tool for the task of creating, manipulating and publishing sites on the Web.

One of the main issues of this project is to study different ways and tools to create a prototype of a software capable of extracting information from one published site and reuse this data as input to a Content Management System in order to enable it to work with information that is already published on the Web rather than having to create the site from the scratch using the tool.

---

1A Content Management System is a software used to ease the publishing work of web sites. It is often composed of two elements, the CMA (Content Management Application) which is basically the interface where the user makes the inputs, create new pages and manage existing ones. The other element is the CDA (Content Delivery Application) which processes the creations, changes and exclusions made by the user and updates the site.
Dotify AB\(^2\) is an IT-consulting company that develops its own products and offers consultant services. Dotify AB has developed a product called *Dotify WMS (Web Management System)* (Dotify AB 2001) that works as a CMS.

*Dotify WMS* has features that simplify the publishing tasks of web pages as, for example, linking, translating and copying information between different sites. The system is object oriented and every object has its foundation in XML and XSL. The installation is made on a central server and the system can then be reached through a common web browser. The system is appropriated to create web structures for big organizations with several sites as well as for smaller intranet publications.

### 2.2 Motivations

*Dotify WMS* is designed to handle, publish and update sites that were created with *Dotify WMS*. One weakness in the system is that it cannot be used to import sites that already are published and then modify, restructure and publish the sites again.

For this task, the system needs an importer module to scan arbitrary sites and reuse the information found in the site so that the *Dotify WMS* can be used as the Content Management System by the sites company.

The lack of the importer function in the system is the issue this project intends to solve. Dotify AB has already had a student, Johan Haldin from the Uppsala University, during 2003 that started a project to create an importer module to *Dotify WMS*. Johan Haldin’s project (Haldin 2003) became limited to fetching all the pages, files, pictures, folder etc. from an arbitrary site and saving them in a local database and file system. The database works only as a storage place for the data, with the system not using any features of the database manager.

Despite the limitations made by Johan Haldin the prototype fully accomplishes the objective of fetching and storing data from the Internet. The idea is to use Johan Haldin’s prototype as starting point for this project, since it has been accepted as a fully functional system by Dotify AB. Johan Haldin’s project is also approved by the Uppsala University.

Thus, the fetching of input data for the prototype occurs from the database instead of directly from the Web.

### 2.3 System Overview

The figure 2.1 shows the arrangement of the system. The left corner represents arbitrary sites on the Web where the home pages are published. The extraction process (detailed description in section 3.1) makes exhaustive requests to the site and demands all objects like HTML pages, images, scripts, pictures, files etc. These objects are then stored locally on the hard disk and in a local Microsoft SQL database.

The steps inside the box marked “core part” constitute the crux of this work. The details about these steps are in chapter 3.

#### 2.3.1 Main Features

The central purpose this project aims to unfold is to analyze the procedures needed to establish methods apt to:

- identify,
- convert.

\(^{2}\)Dotify AB - http://www.dotify.com
2.4 Problem Definition

An alternative way to settle the goals for this project is by observing the implications embedded in the assertions made in section 2.3.1. Concerning this project the statements pragmatically involve the investigation of the possibilities to transform information that already is published in the Internet to another format that can be used in a Content Management System, in this case, the Dotify WMS. The data format to be used by Dotify WMS is not yet defined so the goal is to find relevant data among the pages in an arbitrary site. As XML seems to be the most suitable format some studies have been made to create XML files from the information originally found in HTML format.

In order to accomplish the main goal, a few different steps and problems can be defined.

- Extracting Data from Internet
- Parsing HTML Code
- Structuring HTML Objects
- Browsing the Structured Objects
- Search Algorithms (string match)
Chapter 2. Introduction

– String Search
– Regular Expressions

• Search Algorithms (tree nodes)
• Exporting Data

2.5 Delimitation

Several problems have been faced during the development of the project of which some will be introduced in this section. Problems that many times end up in concerns that do not regard this project, like remedying, cleaning up and/or regulating HTML and script code. Although the restrictions presented in this section are plausible, they ought to be deeper discussed in order to achieve a full featured system.

• Problems related to HTML pages.
• Problems related to the presentation of the pages.
• Nested objects in HTML.
• The prototype development.
• The data format for extraction.

2.5.1 The HTML Pages Code

A very common problem with tools that parse HTML code is the fact that many pages on the Web actually are not W3C\(^3\) compliant. The W3C’s specifications for HTML pages provide web pages validation\(^4\) in order to ensure the pages are correctly coded, but unquestionably most of the pages on the Web are not properly coded according to those stipulations.

Usually browsers like Internet Explorer and Netscape have means to bypass rendering problems of improperly coded pages and tolerate many kinds of tag mismatches without compromising the appearance of the page in the browser.

**Restrictions**

The attempts to work with such faulty pages showed abnormal behavior of discrepant forms. These transgressions are even more aggravating when parsing mismatched table tags. In the prototype developed for the purpose of tests merely a quick check is made in order to control the balance of opening and closing tags. The section 2.5.3 elucidates the problem in details.

Even though highly desirable, this sort of functionality is not going to be provided in the prototype of this project. Flawed pages are just left apart without further correction and processing.

2.5.2 Browsing the Objects

Nowadays, in order to create dynamic sites (instead of static) the use of scripts in Web pages are becoming more and more frequent.

\(^3\)The World Wide Web Consortium
\(^4\)http://validator.w3.org/
2.5. Delimitation

In this project, the scripts in a page are detected and managed by the system as any other HTML object defined in the rules used to identify HTML tags. There is no other crucial aspect referring to scripts, besides the fact that when rendering objects in the prototype’s browser, the objects rendered derive the string (representing the object itself) from their own class plus from all their subtrees. So, if in one part the code points to a script declared somewhere else in their parent class, that script will not work properly.

Figure 2.2 depicts an example, where the HTML object `td` is selected. In the browser (though not showed in the picture) the object is presented only by it’s own (HTML) code and by the code of it’s children (three tables in the picture). If e.g. in any of these tables there is a function call to a function stated in the `body` node, that script fails to execute.

There is no other consequence than an exception (and an annoying warning) caused by the browser. For the matter of the purposes of the project no inconvenience is caused.

Restrictions

No attention has been paid to that exception inferred by the browser class. Certainly there are many ways to avoid the problem, but it has been seen as a minor matter.

2.5.3 Nested HTML Objects

The problem with tables is that mismatched tags imply inadequate parsing of the page, besides the fact that there are a number of ways to create (apparently) the same table. A more peculiar problem is the sequence the parsing process has to follow. It is not a simple matter of finding start `<table ...>` and end `</table>` tags. The likelihood of nesting objects inside objects suggests the necessity of finding the right pair of tags.

The examples in figure 2.3 shows the awkwardness of distinguishing the tables.

The Example 1 in figure 2.3 is obviously wrong while the Example 2 is correct. Section 3.2 characterizes more about the problem with nested objects and explains the methods for instantiating and allocating them as nodes in a tree which has `html` objects as root.

---

5 `td` stands for “table data”
6 A set of objects of graduated size that can be stacked together, each fitting within the one immediately larger. (objects inside objects).
Restrictions

The HTML pages processed by the prototype requires proper HTML code. Tables without correct start and/or end tags will not work properly in the parsing process. Again, no intentions to repair the disagreement of misplaced or absent tags are made.

2.5.4  The Prototype Development

As most likely with every prototype, the development of this project’s prototype was aimed essentially at the accomplishment of satisfactory results by the system. The main considerations regard precision, regularity and accuracy of the system. The meaning of the prototype is to show ways to achieve the desired results.

Restrictions

With these aspects in mind, the restrictions mentioned here concern the memory usage and performance of the prototype, which nonetheless must be taken in consideration in future development of a product.

The prototype developed still has many breaches where it is possible to speed up the general performance of the system and to release memory resources, but that is left to an eventual new version of the system. The appendix A - The Prototype - tells where (and sometimes why) some improvements can be made.

2.5.5  The Final Result

Considering that the class definitions for the objects that are supposed to be extracted to the Dotity WMS are not defined, it is beyond the scope of this project to compute further steps after the matching HTML objects are identified.

Restrictions

The final output from the prototype is still formatted as Web data (HTML code, scripts, etc.).
Chapter 3

Methods

This chapter converges in the methods applied to the specific actions named in section 2.4. Although each action has its own and well defined start and end entries, the steps are noticeably dependent on each of the antecedent step(s) to perform the whole task. Of course it would be achievable to exchange one or more processes, but in this work, no ambitions were made to make isolate modules of these processes.

A description of the functionality of the prototype in appendix A will show where and when the individual operations are called.

3.1 Extracting Data from Internet

Data extraction from the Web, in form of HTML pages, scripts, files, folders and pictures, is not a difficult issue. A prototype (Haldin 2003) of this phase has already been presented by another student and this prototype has been used as the mechanism for fetching data from the Web.

The approach used here is, if not simple, at least straightforward. Given an web site address (URL), the algorithm finds all the links and references in the start page, puts them in a queue and continues with the addresses in the queue, making new requests to the web server, but only if the addresses are located inside the domain of the site, leaving outside references as end points. All textual data is stored in a database, while other files like image, pictures, pdf files are stored in the local file system. The figure 3.1 gives an overview of the functioning of Haldin’s prototype.

The proceedings are shown by the numbers inside the squares in the picture. Each step is described below.

• 1 When the address (URL) of a given site is provided (www.dotify.com in the example) the first step puts it in a queue. When the queue is checked the address is found and passed to a function that makes the request.

• 2 The request function then removes the address, dequeuing the line, and prepares the request.

• 3 The request is sent through the Internet to the specified address.

• 4 The web server (at www.dotify.com) replies and sends back the site’s first response.

• 5 The response received is then analyzed and the links and pointers to other objects like script files, images and pictures are identified. These pointers (that in fact are (URL) addresses to objects) are placed in the queue for later requests.
When these responses and objects are returned by the web server the procedure decides where to save them, depending on the kind of data. Textual data like HTML pages and scripts are sent to the database. Other files are saved in the local file system.

As Haldin’s prototype saves all data locally, the approach used in this work is to retrieve information from the database and the file system. The transparency of this process seems too obvious to be commented in this rapport.

### 3.2 Parsing HTML Code

Since the whole pages are treated as single and long strings and due to it’s flexibility and efficiency when used in text processing, the rules for parsing Web data in this project were written as regular expressions. The regular expressions work similar to an embedded programming language that permits the description and parsing of the HTML source code.

Microsoft’s .NET Framework offers powerful classes for developing applications that use regular expressions (Friedl 2002). The regex tools in the .NET Framework are built with a traditional NFA\(^1\) regex engine, which implies greater control possibilities of the matches\(^2\).

The regular expressions for matching HTML tags (or objects) in this project are written in a XML file that is read only once, when the parsing process is initiated. The XML file contains several regular expressions for many different kinds of HTML tags. Below is an example of the regular expression used to identify the Header data in a HTML code:

```xml
<identifyHead>
  <id>head</id>
  <regexp><![CDATA[<head>(?<content>.+?)</head>]]></regexp>
</identifyHead>
```

In this particular case the regular expression itself is contained between the `[CDATA[ ... ]]-tags and it says: “capture everything between `<head>` and `</head>` and put in a variable called content”.

---

\(^{1}\)NFA is a kind of Finite-State Automata (FSA) meaning Non-deterministic Finite Automata (Harel 1997) (Parsons 1992), (Aho et al. 1986)

\(^{2}\)In opposite to the DFA (Deterministic Finite Automata) engine that keeps track of all possible matches simultaneously, ignoring different representations of the regular expression.
3.2. Parsing HTML Code

Originally the idea was to create as many rules for extractions as there are HTML objects, though during the tests done with the prototype it became clear that with just a few regular expressions for identification of the most common HTML objects a good result could be achieved.

A description of the rules in the XML file containing the regular expressions used in the prototype, is presented in appendix B.1.

The process of parsing a HTML page works as shown in figure 3.2. The tag identification starts with the HTML code as input. The class Tag_Identifier reads in all the regular expressions in the XML file and searches for one kind of object at time\(^3\). In the example in figure 3.2, the process finds the start (<head>) and end (</head>) tag for the head object (1). The substring that describes the head object is collected by the regular expression and the object is instantiated. The process then saves the object in a set (2). When the head object is instantiated it also checks for any eventual sub node (3). In the example, the set in the class Tag_Identifier already contains a title object which the head object identified as one of it’s child node. The head object then requests it’s child node from the set. The title object is returned to the head object (4) as a sub-node.

The process continues in the same way until all the regular expressions have been applied and no more objects can be identified.

The call to that function returns when an object of type html is found, as it is the root node of the tree. It is the html object that is returned to the caller function.

3.2.1 Parsing HTML Tables

Another considerable issue about the process of parsing HTML code is how tables are treated. As told in section 2.5.3 the pages containing tables must have properly matched pairs of tags. Other HTML objects like table data (td) and table row (tr) have an optional end tag\(^4\). Concerning this project these objects are caught by regular expressions only when their respective closing tags are present. Otherwise these objects are treated as part of the container object (which would be their parent(s)).

With HTML objects the consequences of having objects nested inside other objects is that the correct pair of opening and closing tags have to be identified. That is true for every (HTML) object that spans over it's doublet of tags (e.g., <title> and </title>, <html> and </html>) and even for tables.

With most objects, it is fairly straightforward to find the closing tag as it arises as the first occurrence after the opening one. At first glance it may appear to be the case in every context, but the situation in the examples in figure 2.3 proves the opposite. However, what makes table objects special during the parsing process, is that the action must be inverted. That is to say that the approach is to find the closing tag first and subsequently, catch the first appearance of the opening tag antecedent the closing tag.

\(^3\)It identifies all occurrences of the same object type before starting applying the next regular expression to identify the next kind of object.

\(^4\)According to the specifications of the W3C (see http://www.w3.org/TR/html4/struct/tables.html#h-11.2.6)
Chapter 3. Methods

... <table ...> TABLE_A </table> ...<table ...> TABLE_B </table>...

This concept works fine once the tables are flat as the example code above, i.e., when no table(s) lay(s) inside other table(s).

In the figure 3.3 below, the code illustrates that parsing nested tables carelessly leads to a disarranged result. Applying the simple approach of finding pairs of tags closest to each other can lead to faulty conclusions. The numbers inside circles show the sequence the tables would be found and instantiated\(^5\).

\[\begin{array}{c}
\quad 3 \\
\quad 2 \\
\quad 1 \\
\end{array}\]

\[\begin{array}{c}
\quad \text{... <table ...> TABLE_A </table> <table ...> TABLE_B <table ...> TABLE_C </table> </table> ...} \\
\quad \text{Figure 3.3. Erroneous parsing of nested objects.}
\end{array}\]

As tables can not be truncated as in figure 3.3, in order to decompose nested objects a complementary method has to be applied. What this method has to do is to start from the innermost table and then work it’s way up to the outer table containing all the others.

For this purpose, recursion is the right answer. As shown in figure 3.4, the table lying deepest inside the cluster (table_1) must be handled first, then table_2 and so on until the algorithm reaches the table that is the container of all others (table_4). Doing it by recursion is just a matter of finding the first occurrence of the closing tag (</table_1> in the example) and then the opening tag.

\[\begin{array}{c}
\quad \text{<table_4> <table_3> <table_2> <table_1>...</table_1> <table_2> <table_3> <table_4> } \\
\quad \text{Figure 3.4. Correct parsing of nested objects.}
\end{array}\]

The pseudo algorithm below illustrates how the flattening process works. Every time a new table is found the string representing it is replaced by an identificator. All the tables (or other objects) inside table(s) end up in a single identificator string.

**Pseudo algorithm**

```
input: html_code
while html_code contains <table>
    end ← first occurrence of </table>
    start ← last occurrence of <table> before end
    tmpString ← string between start and end
    initialize table object with tmpString
    replace tmpString inside html_code with some id
    continue tmpString inside html_code with some id
return
```

The modus operandi for the whole parsing process can be divided in two main methods. The first

\(^5\)Regardless if the approach is to first find opening or closing tags, the order the tables would be found would still be the same.
3.4. The Presentation of the Object Tree

step de-compounds objects that are nested inside tables, replacing each table object by an identifier. That collapses the implicit hierarchical anatomy of HTML code that, instead, is relocated in the tree structure. Thereafter the process continues with the flattened objects that remain in the HTML code. The major advantage of arranging the objects this way is that the tree structure keeps the the relationships between the objects on the page.

In fact, the html node is the only object returned after the parsing process is finished. The entire creation of the subtree derived from the html object is done by each and every node themselves.

3.3 Structuring HTML Objects

Considering it’s implicit multi level structure, the most natural way to structure the HTML objects found during the parsing process is by creating tree structures with the nodes representing the HTML objects, since HTML objects have a hierarchical relation between themselves.

Figure 3.5 shows an example of the result of the parsing process of a (very) simple HTML page that generates a tree structure. There is no doubt that the average size of HTML pages varies significantly, but generally the number of nodes or objects extracted from typical HTML pages easily exceeds one hundred.

During the parsing process, each new object or node is responsible for the instantiation of it’s sub nodes by consecutively making calls to the Tag_Identifier class depicted in figure 3.2. It is a quite simple recursive process supported by a static class that simply recognizes, instantiates and stores the objects in a set. Every time a new object is instantiated it also checks if it has sub nodes, in this case it asks the static class, which returns the object from the set or dives into a new recursion to repeat the recognition process for new HTML objects. The level of recursion is directly related to the depth of the tree created by the parsing process.

The recursivity of the parsing process maintains the hierarchy of the objects, since whenever a new object is found a new node is set to the tree.

During the “flourishing” of the tree the steps described in this section and in section 3.2 (Parsing HTML Code) work together, almost simultaneously, since for each new HTML object detected during the parsing process a new node is added to the tree. When the html object is returned and the last object instantiated the resulting tree structure can be browsed and rendered.

3.4 The Presentation of the Object Tree

Microsoft .NET Framework offers (among many) an efficacious class, TreeView, that supports the rendering of tree structures and nodes. This TreeView is used as the implementation class of the tree structure mentioned before. When the parsing, identification and structuring processes finish, the objects can be viewed in a simple browser. This simple browser is not meant to operate as a

---

6 Except for the leaves (or ending nodes), which of course, do not have any sub-nodes.
7 the Tag_Identifier class named in section 3.2
navigator since no links can be followed as in normal browsers. It is just a tool for rendering Web pages.

Each node, when selected, returns the string that characterizes it and all its child nodes. The simple browser can then show the contents for that object. The screen shot example in figure 3.6 shows how selecting the root node (the html object) the whole page is shown in the browser. Figure 3.6 shows only the part of the prototype used to navigate between the nodes (TreeView) and the simple browser.

The idea is to allow the user to select nodes and objects that are to be compared and/or found in the other pages of the site. Other objects can be showed in the browser by expanding and selecting other nodes of the tree. By moving across the tree the user can select specific parts of a Web page. These steps will be described with more details in the appendix A - The Prototype.

3.5 Detecting HTML Objects

The bottom line of parsing HTML pages, structuring objects and presenting the hierarchical tree is to allow the selection of objects for further detection in other HTML pages inside the site inspected. Although all the steps described so far are essential to achieve the goals of the project’s purposes, the detection process, together with the parsing process, are the core of this work and will be described in greater details.

The basic idea was to use the string of characters that describes the template objects to create several search parameters that would be applied to the other pages. The searches that scored enough points would then be selected, parsed and presented as final result, ready to be exported to the Dotify WMS.

Different approaches have been tried in order to locate settled parts of a HTML page, which we may call “the template”, since they work as a patterning model in the searches. Rather sublime variations of these templates lie inside the other pages and the idea is to find them in order to export these objects for later remodeling in the Dotify WMS. Further, the node being compared to that template will be called “the comparing node” or “the comparing tree”.

The methods attempted varies from string search to structure (subtree) comparison. The following list shows the approaches scrutinized for this task.

- Appearance search
  - String’s methods
  - Regular expressions

- Regular expressions
  - Regular expressions for HTML tags
3.5. Detecting HTML Objects

- Regular expressions with minimal matches
- Regular expressions with maximal matches

- Subtree/Structure comparison
  - Scoring system
- Subtree/Structure comparison & Regular expression

3.5.1 Appearance Search

The assumption made for this approach was that the objects to be found would present the same characteristics as the template. Characteristics like background color, cellspacing, cellpadding etc., should be pretty much the same on the template object as on the other pages that actually hold similar object(s). The idea was to gather a stack of the properties from the template and then find matching substrings in the other texts. The substring containing most of the comparison properties should be the string declaring the searched object equivalent with the template object.

Using methods from the String class in .Net Framework like IndexOf, LastIndeOf, Equals etc. to compare and find substrings were, from the beginning, considered quite inclined to fail, but no condemnation were made before some investigations. The reasons for the suspicion were raised considering the characteristics of the searches. The String class' methods are simply not adequate for text searching, since, most often, the comparisons are made between a fairly long string and a whole Web page.

The results worked much like the witness needed to convict the criminal and actually the methods created with string search in mind were discarded rather quickly, essentially due to the error prone character of indexing of substrings.

Although the failure of the approach with string match using methods from the String class, the idea of considering the appearance of template object persisted.

3.5.2 Appearance Search - Regular Expressions

The use of regular expressions is the most adequate for text processing regarding the tag matching characteristics of the searches applied. Thus regular expressions have been plainly used in the searches. The next effort was to try the same philosophy as above, comparing properties that give particularities to the template object, but this time using regular expressions instead of methods from the String class in the .Net Framework.

The guess here was that avoiding the index picking in the method above better results would be achieved. The idea was to start by collecting the pair of tags surrounding the template object, e.g., <table ... </table>. Then, using regular expressions spot all the sub-strings in the target text that have enclosing tags like the template. With these matches in hand new regular expressions were applied to make specific searches for the property-value arguments associated to the template.

The example below gives a better picture of the property-value arguments that work as a queue. Figure 3.7 shows how the method works. First applying a regular expression to match the surrounding tags and then to match the arguments that give the appearance to the template object.

\[
\begin{align*}
[0]: \text{height}='100\%' \\
[1]: \text{width}='174' \\
[2]: \text{align}=\text{top} \\
[3]: \text{bgColor}='\#eedeee'
\end{align*}
\]

8"We need jurors... who will not convict merely because they are suspicious" (Scott Turow).
Chapter 3. Methods

This approach gave in some cases satisfactory results, but one significant problem was to correctly match the right tag pair. The contexts in which the process gave unsuitable results were far too frequent and even if improving the precision of the regular expressions could work, they often turned out to start missing tags somewhere else.

3.5.3 Searching Embedded HTML-tags

The next attempt was to enhance the aim by fine-tuning the searches. So instead of only seeking for properties inside the embracing tag pair as the method described above in section 3.5.2, the ambition on the new method was to apply the same approach as above, but for each and every object lying inside the tags. In other words, a divide-and-conquer technique should allow a scoring system to collect points for every match, giving the whole search a final score. This final score would then be compared to the other searches and the best one would correspond to the final result.

The method would first match the first surrounding tags, then compare the properties of the target and the template object. Then find the next pair of surrounding tag of the next object and compare it’s properties and so on until the properties of the innermost object have been compared. Actually the results obtained from this method were quite good, but about three to five times more time consuming comparing with the methods described so far.

Besides the tag matchings, this method focused on matching properties from every object inside the container of the others, but executing recursions like this at string level is a pretty heavy engineering, considering the amount of text it is applied to.

So time became the deadly shot for the method described above. Something else must be done.

3.5.4 Minimal Matches

On account of the results obtained with the method above some performance improvements were tried before either conclusively rejecting or admitting the approach.

Since the execution time was the bottle-neck of the approach above, the idea was to find minimal matches and minimize the amount of text to be compared and hope that one of the minimal matches would include all the objects needed to determine the searched area in the HTML page to achieve a good result. The method used to find matches with minimal length was using non-greedy properties of regular expressions.

But seeing that it is difficult to find the exact pair of matching tags in a text (HTML page) without having to parse the whole page, the use of regular expressions with non-greedy properties actually minimizes the chances that the matches would contain all data needed for the result.

And indeed, because the nearly random matches in the approach described in section 3.5.3 the matches were most of the time insufficient. Besides, the non-greedy regular expressions matched only a small part of the desired object, leaving too much data outside.

9Observe that there is no scale to rate or validate the results. One result is either complete and good or it is totally worthless, since the data defining the structure to be exported must represent the template completely, but (most likely) with another content. A missing object would make the result unusable.

10As the matches did not bother to find the correct pair of tags.
3.5. Detecting HTML Objects

3.5.5 Maximal Matches

The next idea was to do the opposite and use maximal matches, i.e., collect as much as possible with (greedy) regular expressions and hope that any eventual superuous information would fall off during the posterior parsing process. Well, even if the trial of collecting maximal matches seems stupid considering that the problem of not finding the correct pair of surrounding tag persisted, this approach turned out to be very useful in judging which pages that really contains the searched objects. Although the method did not manage to entirely execute the task of detecting the correct objects and returning the complete string that renders the object, it became a quite convenient help function to the final method developed to detect similar objects as described below in section 3.5.6.

The final approach must be totally reviewed. It became obvious that trying to handle the implicit hierarchical structure of HTML code with regular expressions and string matches was a too hard, too costly and a too error prone task.

3.5.6 Subtree Comparison

All the tries with the methods described above (from section 3.5.1 to 3.5.5) were made in order to avoid having to parse whole pages in order to detect the interesting parts. As the problem of achieving completeness in the matches persisted, a different approach must be developed and the requirement of not parsing entire pages must be reviewed. Here the meaning of completeness regards the return value of the matches, that in order to be useful, must be:

- **exhaustive** → without any omission of data and
- **atomic** → not containing additional data.

These conditions are much easier fulfilled by gathering the information contained in the parsed pages. Giving up the dilemma of parsing entire pages renders the possibility of tackling the problem, only compromising the total performance with the additional time needed to parse appropriated pages.

Since most of the sites contain huge amounts of HTML pages, parsing and creating tree structures for all of them just to find specific parts, which may be found only in a fraction of them, is not the smartest thing to do. As the maximal matches (section 3.5.5 above) presented good performance, it was used to make a quick check of all the pages in order to show which pages are worth the job of being parsed and structured.

Then, once the pages with the highest chance of containing variations of the template object are selected, the job is passed to the parsing process showed in section 3.2, which then creates the same kind of tree structure as the template. These trees structures are then used to detect similar objects.

The greatest advantage of having the HTML objects structured in a tree is the fact that each object handle it’s pair of initial and final tags, it’s properties and mainly it’s role in the hierarchy of objects. So the comparison process is highly simplified in question of finding the correct pair of tags from each object. Now the hierarchically organized objects themselves handle that kind of information.

Having the tree structure in hand the searches become now even a matter of comparing subtrees. As the user selects the object(s) to be compared, the template for comparison is defined. The algorithm works with one tree at time and instead of searching through the whole tree, the performance of finding the object is highly improved if a quick selection of the objects of same type as the root of the template is done before the application of regular expressions for comparison of properties.

---

11 Which has been the biggest problem with the other approaches.
Normally, variations of the objects selected as templates lie in the same position of the rendered HTML page as in all the other pages that actually contain these variations. The examples in figure 3.8 below shows different HTML pages with the content area being slightly different in each case.

Figure 3.8. Variations of content in three different pages.

It is these content areas in the pages that must be found when the equivalent area in the template object is selected. Besides, the location of the selected template section is even embedded in the tree structure created during the parsing process, which means that the location of the searched node in the tree is also conclusive. Using this information the searches are minimized to the number of objects of the same type as the template’s root that are located at the same distance, having the same kind\(^{12}\) of parents all the way up to the root node (a html node in the treeView). Of course, typically there are a large number of such objects so the search goes further, but now working downward the tree using regular expressions to compare properties of the objects.

Outlining the assertions above:

- the nodes where the searches start are of same type as the root of the template.
- the nodes most predisposed to give the right result lie at the same distance from the root node as the template.
- the parents of the template and the comparing node are (at least partially) of the same type.

Assuming the statements above the steps can be summarized in:

1. array ← collection of the objects of the same type as the template node.
2. for each object in array ← compute the distance to the top node (html object).
3. score ← matching types between each of the searched node’s and the template’s parent.
4. select the best scores.
5. apply the algorithm from section 3.5.5 for each of the nodes with highest score.

\(^{12}\)The “kind” or “type” here refers to the object’s classification according to the rules used to determine the class of the objects found during the parsing process, e.g., html, head, title, table, etc.
3.5. Detecting HTML Objects

The final action uses the string of characters that describes the template objects to create several search parameters that are applied to the other pages. The searches that score highest points are then selected and presented as final result.

**Tree Comparison Analogy**

Figure 3.9 denotes the template node (and its sub-nodes) named in this chapter. This template could e.g., represent the content area mentioned in figure 3.8. The whole section is an arrangement of the properties of the template node, its child-nodes and their sub-nodes, each one acting like a stand-alone HTML object that together compose the content area.

The figure 3.11 is an abstraction of a tree structure of a parsed HTML page (here called comparison tree). The top or root node is the html object. Under the root node there are all the sub-nodes corresponding to the other objects like title, tables, and links.

Observe that the template node in figure 3.9 is not the root node of the whole tree representing the HTML page, but only the root node of the objects to be compared when the algorithm delineated in section 3.5.5 is applied.

The analogy is made between the template subtree and the recently parsed tree (figure 3.11), that here is called comparison tree. The figure 3.10 shows the abstract significance of the nodes in figures 3.9 and 3.11.

- **The root node for respective tree.**
- **Other (unimportant) nodes in the comparison tree.**
- **Building blocks for the template tree.**
- **Content related objects in the template tree (figure 3.9).**
- **Objects working as building blocks for the comparison tree.**
- **Content related objects in the comparison tree (figure 3.11).**

**Figure 3.10.** The abstraction of the different nodes in the template tree (figure 3.9) and in the comparison tree (figure 3.11).

First all the objects from the comparison tree that are of same type as the template node (see figure 3.9) are collected in an array. Then, the distance towards the root node is computed for each object in the array. This scoring function is made in a scoring system that, while moving upwards (in both the template tree and the comparison tree) all the way up to the root (html) nodes, checks
the equality of the nodes’ type. If both are of same type the scoring function computes a number of points for that node; otherwise a withdrawal is made.

The fundamental benefit of this step is to locate nodes in the comparison tree that are located at the same position as the template node correlated to the template tree. The position of the node in the tree tells where the object is rendered in the browsed page.

Each node in the array still holds their subtrees so when the scoring system has computed the scores for every node from the array the nodes that achieved the highest scores are selected for further comparisons. The comparison is now made between each node and the template tree. This time looking down to the child-nodes. The algorithm applied is the one described in section 3.5.5.

The algorithm starts collecting the two strings that define the template object and the object to be compared with the template. The string that describes the template object is divided in substrings which are then put inside regular expressions. The string that represents the comparison object is the target of the regular expressions. For each substring (from the template object) a score is gathered depending on the success of the match. A succession of the scoring system collects points for the matches that succeed.

Finally the node with the highest score is assigned as the node representing the searched area.

3.6 Conclusions and Comments

The approach described in section 3.5.6 - Subtree Comparison - turned out to be the definitive method for searching relevant information on Web pages. The results obtained were considered totally satisfactory considering the correct use of templates. The crucial step is the selection of HTML objects (or nodes in the tree), because the method entirely relies on the selection of appropriate tem-
plates. The assignment of templates must be done by the user, who selects the pages with different layouts. It is the templates that hold information about their location on a Web page and their layout properties like size, cell spacing, color, font, etc. These pages then provide comparison material to the approach.

The choice of the Subtree Comparison as the final approach is based on the results achieved by running the different methods presented in this chapter with the same kind of data (Web pages). The Subtree Comparison approach is actually the only one that worked properly with data from different sites\textsuperscript{13}.

Related to the performance of the prototype there is a deeply relevant aspect that has to be considered, namely that it is a prototype. As mentioned before, performance has not been considered as an essential matter. On the other hand, what has been considered of importance is the regularity and accuracy of the system.

Regarding accuracy, the system is designed to always return a value from the searches. The implication is that one HTML object from each page is always assigned as being the most similar to the template (closest match). If the structured comparison page does not have a sub-tree equivalent (but most likely with a different content) to the template the prototype still has means to assume that some objects resemble the template and will return the one that achieved the highest (although low) score.

Before the Subtree Comparison method is applied, the prototype is designed to run the Maximal Searches algorithm on all the HTML pages from the site. As explained before, the Maximal Searches method creates a set of regular expressions based on the characteristics of the template object and applies these regular expressions in each of the pages in the site. The method also uses a scoring system to evaluate the accuracy of the matches. This scoring system together with a variable indicating mismatch tolerance allows different results depending on how much discordance the user consents.

This step (the Maximal Searches) has only demonstrative significance since it only performs a quick inspection of the pages and shows the pages that are most inclined to contain an object similar to the template. This step is meant to reveal the pages that should be parsed and re-structured for further searches by the Subtree Comparison algorithm. Following this indication the correct sub-tree is accurately returned.

The results obtained using the different methods explained in this chapter lead to the conclusion that it is much easier to locate related objects observing their position in the (implicit) hierarchical structure of HTML code. The former idea of only comparing visual characteristics of (HTML) objects neglects the importance of the surrounding elements\textsuperscript{14}.

Considering that the system is supposed to work with any kind of site, in this project, the use of the Wrapper approach is the most natural, given the demanded generality of the system. The use of regular expressions in order to extract string blocks inside HTML tags is also conclusive, but perhaps not exhaustive, since even other approaches (see section 1) could do the job. The rules for the wrapper (written as regular expressions) can easily be changed (added or removed) in the prototype, since they are not hardcoded in the program code\textsuperscript{15}.

The use of a tree structure to represent HTML objects as nodes is definitely the best way to present the implicitly embedded hierarchical structure of HTML.

\textsuperscript{13}The other methods described in this chapter required (rather small, but) constant changes in order to work adequately with other data than the test data used during their implementation.

\textsuperscript{14}In the Subtree Comparison method this embedded position is reflected as a node in the structured tree.

\textsuperscript{15}As explained before, the regular expressions are located in a separate XML file.
Bibliography


Appendix A

Description of the Prototype

This chapter concerns the prototype written in C# (Liberty 2003) and created during the development of this project. There is no intention to make it work as a complete manual for the prototype. More willingly this chapter regards the grouping of the procedures described in chapter 3 from sections 3.1 to 3.5 in the system in question.

During the development of the prototype several other ideas emerged and while many were discarded some should be considered as potential features for a product. A brainstorm of these ideas will be exhibited in appendix C.1.

A.1 Preliminaries

The prototype that emanated from this project serves mainly as a guide to show a path that can be followed during the development of a full featured system.

The process of parsing HTML code introduced in section 3.2 in page 12 is based on the Wrapper approach presented in section 1.1 page 1.

The steps showed in chapter 3 were deeply discerned before the implementation of the prototype, but even though, during the prototype development many steps and mainly information turned out to be superfluous. E.g., the objects created after the data retrieval from the database, hold many variables that are never used. The memory costs of such unnecessary data are highly considerable. Also, the performance is reduced proportionally to the overhead created.

A.2 Main Features

One of the main concerns of this work, besides the investigation studies of how to export data to Dotify WMS, has been the development of a prototype\(^1\) able to:

1. scan a facultative site,
2. parse it’s pages in order to create a structure with the purpose of enabling the election of individual pieces of objects,
3. search for similar objects or structures in the other pages of that given site,
4. present the results and

\(^1\) (from: http://dictionary.reference.com/search?q=prototype) **prototype**: “An original type, form, or instance serving as a basis or standard for later stages.”
Appendix A. Description of the Prototype

5. export the results in a given format\(^2\).

### A.3 The Application GUI

The prototype of the importer module is intended to work under human supervision, interactively, since the user is supposed to choose objects/nodes in order to perform the searches in the other pages.

The graphical user interface of the prototype is developed only to attend the needs of the sequential steps of the process for extracting HTML objects out of Web pages. The eventual development of a product may treat the interface conception with much greater consideration.

The assumptions made during the coding process of the interface are straightforward and emphasizes functionality. No special care was taken toward the final user since the intention was just to "make things work". The interface of the prototype is neither intuitive, beautiful nor even good.

### A.4 Using the Prototype

This section presents a quick view of the general use of the prototype. Actually the screen-shots are taken from an older version of the prototype, but the functionality to be explained in this section is not subordinated to the graphical interface.

Considering the evolution of the actions that have to be taken in order to fulfill the assignments of this project, the usage of the application prototype is considerably natural.

#### A.4.1 The Start Window

... and parsing the pages

When the prototype is initiated the main window looks like the first picture on the left side of figure A.1. During the initiation it checks the local database and loads the combobox with the names of the sites that already were scanned\(^3\).

![Figure A.1](image-url) Screen-shots of the initial steps taken when the prototype is run.

The next two pictures show the steps taken in order to select a site and to gather data from the local file system and database. When a (HTML) node is selected in the treeview the browser area shows the page as in the rightmost picture in figure A.1. Still in this picture, the option Create Tree in the menu Tools is selected. This step creates the tree structure introduced in sections 3.2 - Parsing HTML Code - and 3.3 - Structuring HTML Objects.

\(^2\)As said before in chapter 2, section 2.4, the formats and classes the exporting process should use and generate were not defined by the time of the writing of this paper, thus the item 5 above became out of the bounds of this project.

\(^3\)Again, the process of fetching Web data from the Internet is done with the prototype created by Haldin (Haldin 2003).
A.4. Using the Prototype

A.4.2 View of the Parsed Page
... and browsing individual nodes

After the parsing process the tree structure is created. Expanding and selecting nodes in the tree enables the user to search for and choose particular objects in the page. In figure A.2 a table row node is selected. When an object is selected the object is rendered in the browser area. By doing so the user can pick the objects considered of interest which then are used as templates for the searches in the other pages of the same site.

A.4.3 Searching for Similar Objects
... or running the Maximal Match Algorithm

When the user has selected the object (node in the tree) that embodies the content of interest this selected object is assigned as the template object that acts as reference in the searches. The next step is to identify the pages that seem to contain similar objects.

The search settings created during the prototype development is displayed in figure A.3. The button referenced with the text “search similar” fires the process for comparing the string that describes the template object with the other Web pages from the site. The process applies the algorithm described in section 3.5.5 - Maximal Matches - and indicates the pages that most likely include objects that are at least approximately equivalent to the template object. The likelihood of the matches correctly indicating similar objects are partly determined by the trackbar denoted with the text “mismatch tolerance”. This trackbar is used to adjust the amount of missing comparison arguments in the searches. Increasing the mismatch tolerance yields additional pages as possible candidates for containing objects similar to the searched template, but at the same time compromising the precision of the matches.

A.4.4 Structuring Candidate Pages
... or parsing candidate pages

Once the previous action has assigned the most predisposed pages for containing variants of the template object, the next step is to create tree structures for each of these pages. In the prototype it is executed by using the button with the text “create tree”. It executes exactly the same kind of process described in section 3.2 - Parsing HTML Code - and in section 3.3 - Structuring HTML.

Footnote: Most of the “search options” are disabled since the tests made have showed them to be inefficient. These disabled options correspond to the other search algorithms described from section 3.5.1 to section 3.5.4 and that were discarded due to their weaknesses.
Appendix A. Description of the Prototype

Objects - except for the fact that the recently created tree structures are placed in the area referenced with the text “parsed candidate pages”.

A.4.5 Finding Similar Objects

The last step is to apply the algorithm delineated in section 3.5.6 - Subtree Comparison. As outlined before, the algorithm finds for each page the objects in the tree structure that mostly correspond to the template object. In figure A.3 the button with the text “find object” calls the algorithm.

Pertaining to the prototype the returned objects from this process are only highlighted by a blinking frame around the object or a blinking background.

A.5 Things to be Improved

- As formerly pointed out, there is the overhead problem spawned by the data overproduction of objects. Many variables and even some classes are avoidable.
- The prototype development highlights the separation of the processes in order to maintain control during the verification of results, but many of these processes should be put together to simplify the user’s life in future versions of a succeeding product.
- The search algorithms can be further improved by avoiding comparison of redundant or superfluous data, mainly from the “leaf-nodes” of the tree.

A.6 Known Problems

- The order the regular expressions are written in the XML file is important in a few cases. One is detecting comments in the HTML source code. The rule that catches comments starting with // must appear somewhere after the rule that identifies comments of type <!- bla, bla, bla ->, since it is possible to have a valid comment that looks like <!- bla, bla, bla, // ->.
- Every html object must lie inside the <html> and </html> tags. It is not unusual to have comments like <!- created by ... -> outside the html tags. These kind of comments will not be a node since they do not lie inside any other object.
- The recognition of raw text inside Web pages’ source code has a lot of picking details, e.g., “>1<” or “>a<” must be treated as special cases.
- The use of threads is not quite consequent. Some threads do not allow the interface to be updated.
- Processes performed by the same thread that controls the GUI can not be interrupted. If the user tries to click or change the focus the rendering of the GUI will occur only when the process has finished.
- The steps described in appendix A - The Prototype - must be followed strictly. There are many “dead ends” if one or more steps are ignored.
- The prototype does not perform any control of which pages that already have been parsed. If the user chooses to create tree structures for additional pages the system just ignores eventually already parsed pages and executes the process all over again.
Appendix B

Regular Expression Rules

B.1 Rules for Detection of HTML Objects

This sections shows what kinds of HTML objects (or tags) that are detected by the prototype. The list below is not whatsoever exhaustive, though, as it is not difficult to insert new rules for other objects it is thus easy to make changes allowing the prototype to recognize other kinds of objects. For the matter of the prototype the list below has been enough to achieve satisfactory results. Adding new objects results in finer granularity of the decomposition of the pages. On the other hand it also affects the general performance since the tree created during the parsing process becomes bigger.
### B.1.1 HTML Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Short description of the rule</th>
<th>opening tag</th>
<th>closing tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;body&gt;</code></td>
<td><code>&lt;/body&gt;</code></td>
</tr>
<tr>
<td>comment 1</td>
<td>Detects commentary text (most used in scripts and among HTML code)</td>
<td><code>&lt;!- -&gt;</code></td>
<td></td>
</tr>
<tr>
<td>comment 2</td>
<td>Recognizes comment of the type /* comment here */</td>
<td><code>/* */</code></td>
<td></td>
</tr>
<tr>
<td>comment 3</td>
<td>Recognizes comments typical in script code</td>
<td><code>// \n (end of line)</code></td>
<td></td>
</tr>
<tr>
<td>head</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;head&gt;</code></td>
<td><code>&lt;/head&gt;</code></td>
</tr>
<tr>
<td>html</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;html&gt;</code></td>
<td><code>&lt;/html&gt;</code></td>
</tr>
<tr>
<td>image</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;img&gt;</code></td>
<td><code>&gt;</code></td>
</tr>
<tr>
<td>link</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;a href&gt;</code></td>
<td><code>&gt;</code></td>
</tr>
<tr>
<td>paragraph</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;p&gt;</code></td>
<td><code>&lt;/p&gt;</code></td>
</tr>
<tr>
<td>script</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;script&gt;</code></td>
<td><code>&lt;/script&gt;</code></td>
</tr>
<tr>
<td>source</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>src=&quot; &quot;</code></td>
<td></td>
</tr>
<tr>
<td>table data 1</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;td&gt;</code></td>
<td><code>&lt;/td&gt;</code></td>
</tr>
<tr>
<td>table data 2</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;td&gt;</code></td>
<td><code>&lt;/td&gt;</code></td>
</tr>
<tr>
<td>table end</td>
<td>Detects the closing tag and saves the substring succeeding it in a variable called content</td>
<td><code>&lt;/table&gt;</code></td>
<td></td>
</tr>
<tr>
<td>table row</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;tr&gt;</code></td>
<td><code>&lt;/tr&gt;</code></td>
</tr>
<tr>
<td>text</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&gt;</code></td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td>title</td>
<td>Takes the substring lying between the opening and closing tag and saves it in a variable called content</td>
<td><code>&lt;title&gt;</code></td>
<td><code>&lt;/title&gt;</code></td>
</tr>
</tbody>
</table>
Appendix C

Brainstorm

C.1 Brainstorm

• The scoring system must be reviewed. Shall points be withdrawn when a match fails? How many points is matching object type worth? Should they be correlated to anything, like the amount of text compared or the number of sub-nodes?

• Think through before raising events. Many of the events in the prototype work only to give a feedback to the user, not actually triggering any data processing.

• It would be useful to create an interface to allow the management of the objects identified in the parsing process.
  – Tags
  – Regex
  – Internal changes

• There are ways to improve the tree comparisons.

• The XML file containing regular expressions could be extended in order to even allocate information about the objects to be instantiated. This could perhaps reduce the number of classes of objects to the minimal.