



ΔEiXTo User Manual

ΔEiXTo V2.8.8.0

1 March 2008

Kostas Ntonas
kntonas@gmail.com

Table of Contents

Table of Contents	i
Table of Figures.....	ii
ΔEiXTo	1
Embedded Web Browser Control.....	2
myDOM Tree Structure.....	2
Tag Filtering	3
Creating the Pattern	4
Configuring the Pattern	7
Following Next Links.....	8
Regular Expressions	9
Extraction Rule Execution.....	11
Virtual Pattern Root.....	14
Successive Optional Nodes	16
Pattern Matching Algorithm.....	17
Auto Mode.....	19
Auto Fill and Submit Form	21
Extraction Rule Maintenance	22
Output to RSS File	23
Sibling Order	24
Statistics.....	25

Table of Figures

Figure 1: DEiXTo GUI	1
Figure 2: Problematic subtree due to tags	3
Figure 3: Subtree after simplification.....	3
Figure 4: Scrollable list with checkboxes for tags to ignore	4
Figure 5: Buttons to rebuild and simplify myDOM.....	4
Figure 6: Highlight browser mode	4
Figure 7: Information about the selected myDOM element	5
Figure 8: Part of a tree extraction rule.....	5
Figure 9: Creation of an extraction rule through myDOM	6
Figure 10: Possible rule node states on the node's local menu	7
Figure 11: Control elements for following links	9
Figure 12: Typical link structure.....	9
Figure 13: Dialog window for entering a regular expression.....	10
Figure 14: Rule execution button	11
Figure 15: Stop execution button	11
Figure 16: Extraction pattern treeview.....	11
Figure 17: Node with a user specified label	12
Figure 18: Results of execution of a sample rule	12
Figure 19: Control elements for output to file.....	13
Figure 20: Part of a sample XML output file	13
Figure 21: Control elements for max number of hits and native URL	14
Figure 22: Buttons for adding and removing pattern node levels	14
Figure 23: Headers from a sample news website.....	15
Figure 24: News header structure.....	15
Figure 25: Pattern for sports news headers	15
Figure 26: Record with optional data fields	16
Figure 27: Rule subtree with successive optional nodes.....	16
Figure 28: Example pattern and target tree	19
Figure 29: Buttons to open and save a wrapper	20
Figure 30: Defining target URLs of a wrapper	20
Figure 31: Button for execution of a loaded wrapper	21
Figure 32: Controls for auto fill and submit form.....	22
Figure 33: Tune button	22
Figure 34: Sub-elements of channel element of RSS output file	23
Figure 35: Assignment of RSS label to a rule node	24
Figure 36: Dialog box for sibling order definition.....	24
Figure 37: Statistics for execution of a sample wrapper	25

ΔEiXTo

ΔEiXTo (or DEiXTo) is a powerful web data extraction tool. It allows users to create highly accurate extraction rules (wrappers), which describe what pieces of data to extract from a web site. It provides a robust arsenal of features and a friendly graphical user interface (GUI) that is used to build, test, fine-tune, save, maintain and execute extraction rules. It achieves high precision and recall in a wide spectrum of cases. This guide describes the functionality of DEiXTo.

A few words about the name of the tool: DEiXTo is an acronym for *Data Extraction Tool*. First of all, Δ is the equivalent of D in Greek. Now, perhaps you are wondering what this ‘i’ character is all about. Well, in Greek ΔEIXTO (pron. dechto) is the imperative form of point at, which is what the DEiXTo user does inside a web browser window when he specifies items of interest by using the mouse.

Figure 1 illustrates the major components of the application window, which will be described in detail in the following sections. Notice that via a horizontal and a vertical splitter, the user can change the size of certain regions.

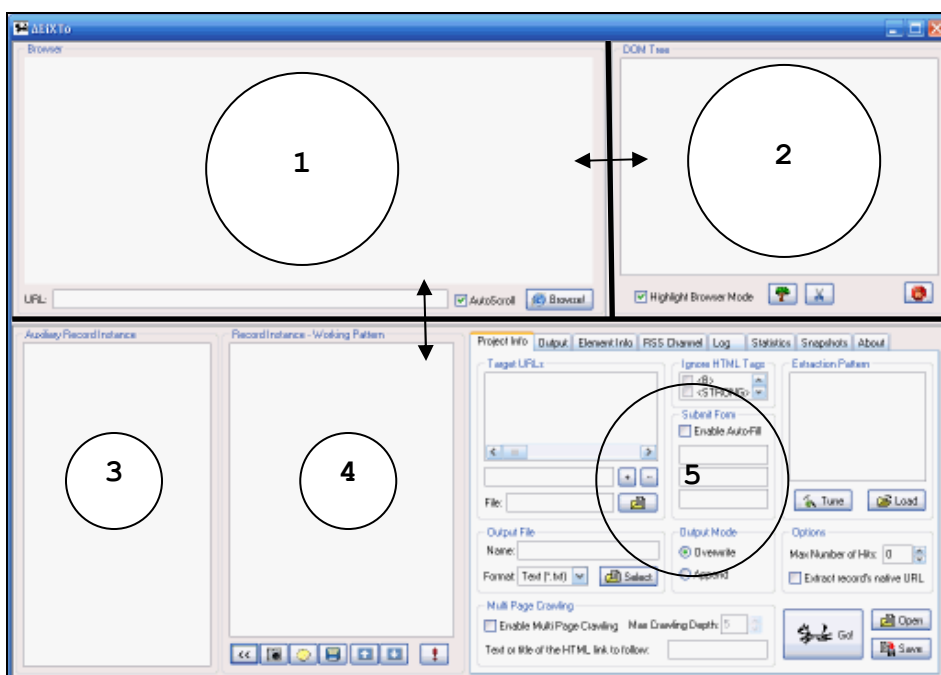


Figure 1: DEiXTo GUI

Embedded Web Browser Control

Probably the most important component of DEiXTo is the embedded web browser control that is located in the region 1 of the window (Figure 1). If the user wishes to extract data from a website, the first thing to do is to enter its *URL* to the address bar and press the ‘Browse’ button. It should be noted that the browser can also retrieve local files from within one’s own computer via the schema *file://path*. In case the browser fails to fetch a page or the timeout, which is set at 5 minutes, elapses, then the appropriate actions are made and relevant error messages are displayed. Moreover, the user can go ‘Back’ and ‘Forward’ via ‘Alt+Left Arrow’ and ‘Alt+Right Arrow’ respectively.

It should be noted that the tool *cannot* deal with *frame* based pages. This is due to the fact that the existence of frames in a page makes its manipulation difficult and requires special treatment because each frame is a different html document. Thankfully, most times this is not the case. That means that the tool’s usefulness is not seriously damaged by not handling such cases.

myDOM Tree Structure

DEiXTo is based on the W3C *Document Object Model* (DOM) Recommendation and thus on the tree representation of an HTML document that reflects its HTML tag hierarchy. DOM is an interface that allows programs and scripts to dynamically access and update the content, structure and style of web documents. It was considered necessary to display the DOM tree of the fetched page inside the application window (Figure 1, region 2). From now on, this tree data structure will be called *myDOM*. The myDOM tree is created when the page is fetched and rendered in the web browser and is built via a classical depth first algorithm and the API that DOM provides.

For each myDOM node, various, useful information is kept. The data which can be extracted is: for <A> elements the `href` attribute, for elements the `src` attribute, for FORM and INPUT elements the `name` attribute, for TEXT nodes their *text content* and for the other html nodes their *inner text*. It is also possible to extract the *source code* of an HTML element.

Tag Filtering

The ability to ignore html nodes while building myDOM proved to be a very helpful utility. This feature was implemented because sometimes certain types of elements encumber the identification of record instances as well as the extraction of the desired data. This will be better understood with the following example. In a typical Google result page there is a serious problem with the bold font of some words. HTML elements force a text string to split in several parts and as a consequence records include structures such as that in Figure 2. Moreover each record has a variable number of words in bold.

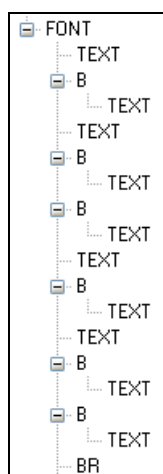


Figure 2: Problematic subtree due to tags

These difficulties can be overcome though. This is done via the simplification method which ignores user specified types of nodes and merges their inner text with the text content of neighbour text nodes. For our example, removing while building myDOM, transforms the subtree above into that represented in Figure 3.

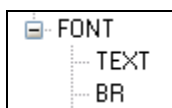


Figure 3: Subtree after simplification

The usefulness of tag filtering is obvious and its advantages are important. The user should first select the tags he wants to ignore on the relevant checkbox list in the *Project Info* tab (Figure 4) in the region 5 of the window (Figure 1).

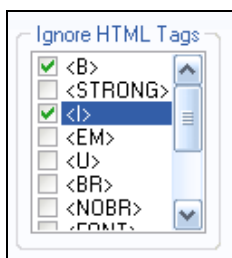


Figure 4: Scrollable list with checkboxes for tags to ignore

What should do next is press the ‘Simplify’ button (Figure 5), which is in region 2 of the window. The user can rebuild the original myDOM tree just by pressing the relevant button next to that of simplification.



Figure 5: Buttons to rebuild and simplify myDOM

Creating the Pattern

Once a page is fetched and it is rendered in the browser window, the user then has to describe the structure of the desired data, thus create a pattern. For this purpose, the browser was enriched with a highlight mode, so that page areas that correspond to visible HTML elements are highlighted when the cursor passes over them. So, if the mouse is over the HTML document and the highlight browser mode is enabled, then the element under mouse is highlighted, if this is possible. The Figure 6 is characteristic.



Figure 6: Highlight browser mode

Moreover, in the *Element Info* tab, in region 5 of Figure 1, useful information is displayed for the selected element (Figure 7), such as the outer HTML of the element, data which can be extracted from it and its absolute path in the document.

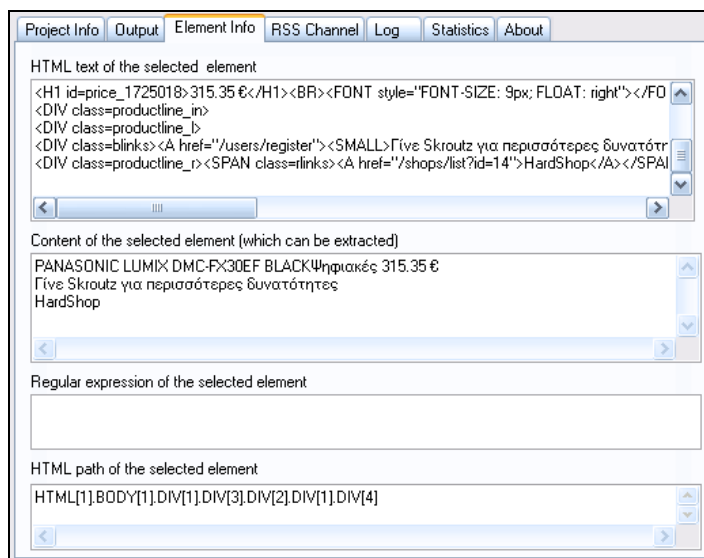


Figure 7: Information about the selected myDOM element

The user can easily and quickly create an instance of data under interest by selecting the relevant option from the popup menu of the HTML element whose subtree corresponds to a representative record instance. Then, a pattern tree is created, which is displayed in the area 4 of the application window. This tree structure is the myDOM subtree rooted at the selected element. Figure 8 shows the pattern that corresponds to the element highlighted in Figure 6. Each node has a name, either a HTML tag or TEXT. The root node is displayed in bold. Note that a pattern by default extracts the contents of the TEXT nodes of each found instance.

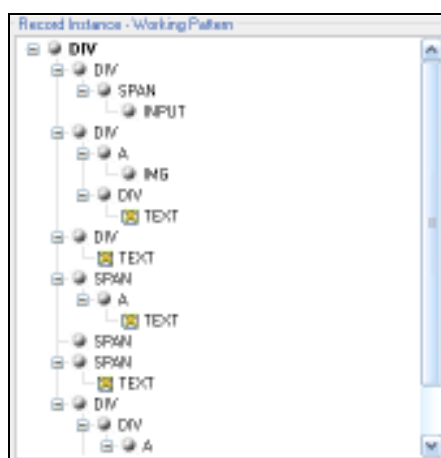


Figure 8: Part of a tree extraction rule

This data structure serves a dual purpose. It is the working pattern and at the same time a record instance. Therefore, when the user selects a rule node, the area of the page that corresponds to the selected node is highlighted. This facilitates the fine tuning of the rule, so as to maximize its efficiency.

In some cases, though, it is not possible to highlight an element. For example, non visible elements cannot be selected via the mouse on the browser window. Then, the user should use myDOM tree to create the pattern. He can select the relevant option from the local menu of the myDOM node under interest. This is shown in Figure 9.

Additionally, a sync mode was implemented between myDOM and the browser. When a myDOM node is selected, the corresponding area in the browser window is highlighted and vice versa.

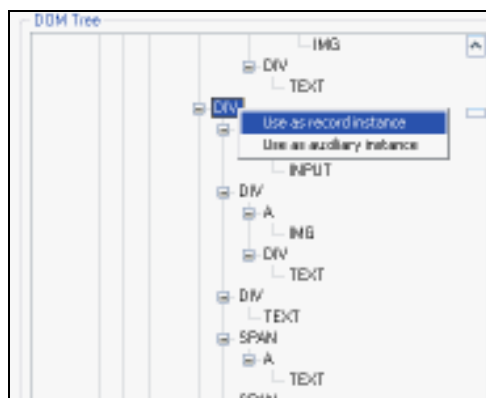


Figure 9: Creation of an extraction rule through myDOM

For cases that it is not possible to highlight an element by using the mouse, the usual practice is to highlight an element near the one we want to “catch”, disable the highlight browser mode through the popup menu and then select the myDOM node that really interests you.

The creation of efficient extraction rules requires the careful selection of a representative record instance, which will be used as a pattern to identify all record instances. A major advantage of DEiXTo is the visualization of the whole wrapper development procedure which makes the creation of even complex extraction rules quite easy and quick.

The tool takes advantage of the fact that semantically related items exhibit consistency in presentation style and subsequently in HTML structure. To minimize the fragility of scraping, it is strongly recommended that the user should use as little

boundary data as possible. Boundary data is the fluff around the actual data one wants.

Configuring the Pattern

Almost always, a just created pattern does not capture all record instances the user wants. In several cases, there are multiple record instances on page which present small or bigger structural variations. These variations are usually due to missing fields. Moreover, the user is interested in specific bits of information that a record instance contains, thus in particular data fields.

DEiXTo allows users to define the role of each pattern node. The user can select among six different node states, each of which expresses whether the node is required or optional in a record instance and whether the user wishes any data from it. The user can select a node state through the local node menu (Figure 10).

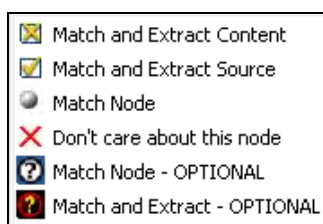






Figure 10: Possible rule node states on the node's local menu

The possible states are:

- `checked`: this node is required in a record instance and the user wants information the node contains. It is an output variable. If it is a `TEXT` node, the data extracted is the *inner text*, for links (`<A>`) the data under interest is the `href` attribute, for images the `src` attribute and for `FORM` and `INPUT` elements their `name` attribute. For the rest HTML nodes their inner text is extracted. In case a node has a regular expression, then the string matched with the target string is extracted. In case parentheses are contained in the regular expression, then the string extracted is the string created by merging the substrings of the string matched with the parts of the target expression in parentheses.
- `checkedSource`: this HTML node is required in a record instance and the user wants the source code of the corresponding element (*outer HTML*). Such a node is an output variable.

-  `grayed`: it is required in a record instance but the user does not want any content from it.
-  `unchecked`: not interested in this node. It could be completely deleted but it is kept for possible future use.
-  `grayed_implied`: this node is optional in an instance and the user does not care to extract anything from it. Consequently, if this node has children, its subtree is optional independently from the states of the rest nodes belonging to the subtree.
-  `checked_implied`: this node is optional in an instance but if it matches with a myDOM node, then its data is extracted as described in checked state. It is an output variable. If this node has children, its subtree is optional independently from the states of the rest nodes belonging to the subtree.

The user can permanently delete a tree node and subsequently its subtree. As the user modifies the pattern, there is the ability to keep snapshots of the current working pattern, which are kept in the *Snapshots* tab. Of course, the user can restore a snapshot and make it a working pattern through its local menu. When the user removes a node, then automatically a snapshot is created.

As discussed before, it is not that simple to locate all record instances on the first shot, so there is a need to configure the pattern and carefully select node states. Very helpful is the auxiliary tree structure in which the user can put a record instance that the pattern missed. This is performed the same way as the pattern is created. This structure is exactly next to the working pattern, so there can be direct side-by-side comparison between the two trees and thus the user can find out why the pattern did not capture the specific instance. In conclusion, this auxiliary structure facilitates the creation and tuning of well engineered extraction rules.

Following Next Links

A typical result page of a search engine or a price comparison engine contains multiple record instances. Very often, the number of search results is large, therefore the records span among multiple pages. The wrappers built with DEiXTo support performing a sequence of page fetches via following ‘Next’ links. Actually, a

wrapper visits all the target pages and gathers all the record instances found on them. The mechanism DEiXTo uses is quite simple but efficient for most cases. The user can enable multiple pages crawling just by checking the relevant checkbox in Project Info tab (Figure 11). DEiXTo identifies the link to follow by using its inner text or its title attribute.



Figure 11: Control elements for following links

The user can enter the name or the title or part (prefix) of it, so as to recognize the desired link among others. The comparison between the string entered by user and the inner text or title of a link is case insensitive. The user can also define the maximum crawling depth, which expresses how many successive pages will be visited at most.

Regular Expressions

Several times, it is very useful to define constraints upon the content of some pattern nodes, so as to ease the location of the desired records. For example, the user can define that a myDOM node, in order to match with a pattern node, should begin with a prefix or contain a specified string. Other times, the user wants to isolate a part of the text contained in a node. These can be achieved via regular expressions. A regular expression is a template to be matched against a string. To better understand the usefulness of regular expressions in DEiXTo, two simple examples follow. Suppose that a user wishes the extraction of the `href` attribute of a 'Next' link. Given that most links have exactly the same structure, the pattern in Figure 12 is not sufficient as it returns almost all links of the page.

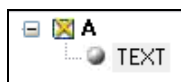


Figure 12: Typical link structure

If the user, however, enters a regular expression for the TEXT node, such as 'Next', then the pattern returns only the desired URL.

Another example that highlights the use of regular expressions derives from cases where the user wants to isolate specific parts of text data. Let's assume that the user wishes the integer value of a string 'from \$249.98' contained in a TEXT node. What he should do is enter a regular expression such as `\$ (\d+)`.

It should be noted that the use of regular expressions applies to both TEXT as well as HTML pattern nodes. The regular expression assignment is realized via the relevant option in the local node menu. In the window that opens, the user can select a pre-built regular expression or enter a new one (Figure 13). To isolate one or more parts of a target string, the user should use parentheses. There is also the ability to evaluate inversely a regular expression, thus using the *not* operator of the given regular expression. For inverse evaluation, the user should check the relevant checkbox. The nodes having a regular expression are displayed underlined. To remove a regular expression and restore a node to its initial state, the user should select the relevant node menu option.

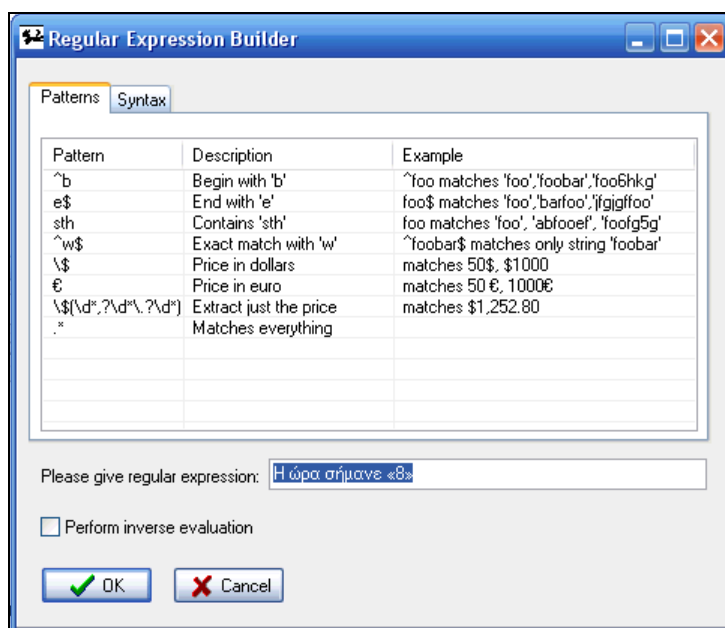


Figure 13: Dialog window for entering a regular expression

Note that regular expressions provide a mechanism for partial mathematical constraints. For example the expression `[7-9]\d\d` matches all integer values ranging from 700 to 999. This could be really helpful in cases with product pages.

Extraction Rule Execution

To execute an extraction rule on the fetched page, the user should press the button displayed in Figure 14, located in the region 4 of the application.



Figure 14: Rule execution button

In some cases, usually when the execution includes a sequence of several page fetches, it is useful to be able to stop the execution. This is done by pressing the button illustrated in Figure 15 in region 2 of the application.



Figure 15: Stop execution button

When the user commands execution, a copy of the rule is created, without the unchecked nodes, and represented in the tree structure of the *Project Info* tab (Figure 16). This copy, whose nodes are empty of data, constitutes the pattern. Specifically, a pattern match effort is conducted on myDOM tree as to identify record instances. For this purpose all myDOM nodes are examined against the pattern.

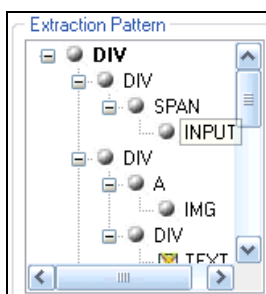


Figure 16: Extraction pattern treeview

If the multiple pages crawling mode is enabled, the procedure continues with next pages so as to gather all record instances. The pattern matching algorithm is described thoroughly later. When a match is found, which means a myDOM subtree matched with the pattern, then a part of its data is captured by the output variables of the rule (the nodes in *checked* or *checkedSource* or *checked_implied* state). The default variable names are VARX (e.g. VAR1, VAR2, VAR3, etc). At this point, the executor collects the values of the output variables and creates an output record. In case a myDOM subtree fails to match, the current contents of the pattern are

discarded and a new effort begins with the next myDOM node. The pattern matching procedure terminates when all myDOM nodes are checked against the pattern. The output results are printed in a list component in the *Output* tab, which is in region 5. The number of its columns is equal to the number of output variables and the number of rows is equal to the size of the result set. The column names are the same as the output variable names, thus VARX by default. However, the user can change the label of an output variable through the local pattern node menu and assign one of his own, thus providing semantics for the data extracted. The label entered by the user is combined with the node name and the character ‘:’ is used as a delimiter, as illustrated in Figure 17. Figure 18 displays sample output results of an execution of an extraction rule for a price comparison engine.

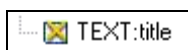


Figure 17: Node with a user specified label

Moreover, the selection of a record in the output list, highlights, if possible, the corresponding record instance on the browser. This is useful for the location of record instances that the pattern missed. Also, double click on an output record opens the page from which it was extracted in a new Internet Explorer windows. This is meaningful when the wrapper executes the extraction rule on several pages and subsequently the output records derive from many different addresses. This facilitates the result verification.

model	price	shop
Kodak EasyShare V610	266.00 €	Katerelos
Nikon D50 6.0	420.00 €	Adorama
OLYMPUS [tmju:] 700 μούρη Παραδιδ...	198.00 €	Pixmania (.fr)
CANON EOS 400D +	715.00 €	Asikidis
Canon PowerShot S3 IS - NEW	369.00 €	Katerelos
SONY Cyber-shot DSC-H5 μούρη Παρα...	382.00 €	Pixmania (.fr)
PANASONIC Lumix DMC-FZ7 μούρη Παρ...	306.00 €	Pixmania (.fr)
Ψηφιακή φωτογραφική μηχανή - Penta...	950.51 €	Megamarket
Nikon D80 Body	869.00 €	Katerelos
Sony DSC-H2	317.00 €	Katerelos
CANON EOS 400D + φακός EF-S 18-55 ...	735.00 €	Pixmania (.fr)
Konica Minolta AUTO METER V F MET...	334.00 €	Adorama
Olympus FE 180 NEW	155.00 €	Katerelos
Nikon D40 Set AF-S DX 18-55/3.5-5....	598.00 €	Technixx.gr
Nikon D80 dSLR (10.2 MP) + Φακός (...	1099.00 €	Πλαίσιο
JVC Everio GZ-MG505 3xCCD	929.00 €	Πλαίσιο
Pentax 67 II AE PRISM FINDER 67 S...	485.00 €	Adorama
Canon PowerShot A640 -NEW-	320.00 €	Katerelos
SONY Alpha DSLR-A100 - Μαύρη Παρα...	682.00 €	Pixmania (.fr)
Sony DCR-HC23E βιντεοκάμερα MiniDV...	273.00 €	Net-electric

Extraction Completed: 20 results!

Figure 18: Results of execution of a sample rule

The output results can be exported to a file. The supported formats are: *tab delimited text*, *XML* and *RSS*. The two latter make use of the output variable names, thus the semantic labels entered by user. So, each label given by the user is used as an XML element type. Since XML has become the lingua franca of the Web, data extracted with DEiXTo and stored in XML can be manipulated and processed in various interesting and meaningful ways. To export extracted data to a file, the user should first select format, name (absolute or relative file path) and mode (overwrite or append) and then execute the rule. The relevant controls are in *Project Info* tab and illustrated in Figure 19.

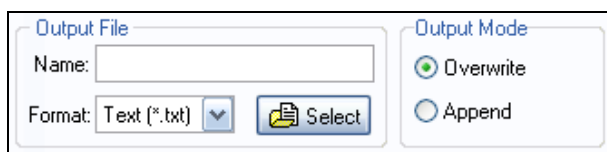


Figure 19: Control elements for output to file

It should be noted that the output files have utf8 encoding without regard to target pages' encoding. If the user selected for example to extract the results of Figure 18 to an XML file, then a file would be produced like that in Figure 20.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <WrapperResults>
- <item>
  <model>Kodak EasyShare V610</model>
  <price>266.00 €</price>
  <shop>Katerelos</shop>
</item>
- <item>
  <model>Nikon D50 6.0</model>
  <price>420.00 €</price>
  <shop>Adorama</shop>
</item>
```

Figure 20: Part of a sample XML output file

In cases of tab delimited or XML output file, there is the capability to extract also the native URL of each record. This can be achieved via the relevant checkbox in *Project Info*. Moreover, the user can define max number of results, which can be used as a termination condition of the pattern matching algorithm. The relevant control elements are shown in Figure 21.



Figure 21: Control elements for max number of hits and native URL

During rule execution, the application is in running mode and all browser events caused by user are disabled until the execution completes. For instance, the user cannot follow a hyperlink; neither can display a popup menu. This is useful for cases including following ‘Next’ links and is done to guarantee smooth execution. If not in running mode, the user can use the embedded browser as usual.

Virtual Pattern Root

In certain cases, the pattern structure can be quite simple and thus wrongfully returns too many results because this structure is very popular in page. This means that the pattern should be stricter, thus some constraints should be defined. Regular expressions are really helpful but sometime they are simply not adequate. To solve this problem, there is a need to describe the environment (or neighborhood) of the pattern root node.

In DEiXTo this is achieved via inserting in the pattern tree some direct ancestors (father, father of father, etc) of the current pattern root and perhaps adding also their siblings. The user can ascend and descend node levels with the buttons shown in Figure 22 that are located in region 4 of the window. The user can also add siblings to an ancestor node of the initial root through the local node menu. Note that the insertion of a sibling node, inserts its entire subtree.



Figure 22: Buttons for adding and removing pattern node levels

For instance, in a sample news website (Figure 23), the headers have exactly the same presentation style and are organized by category in tables. Suppose that the user desires the sport news (‘ΑΘΛΗΤΙΣΜΟΣ’ in Greek).

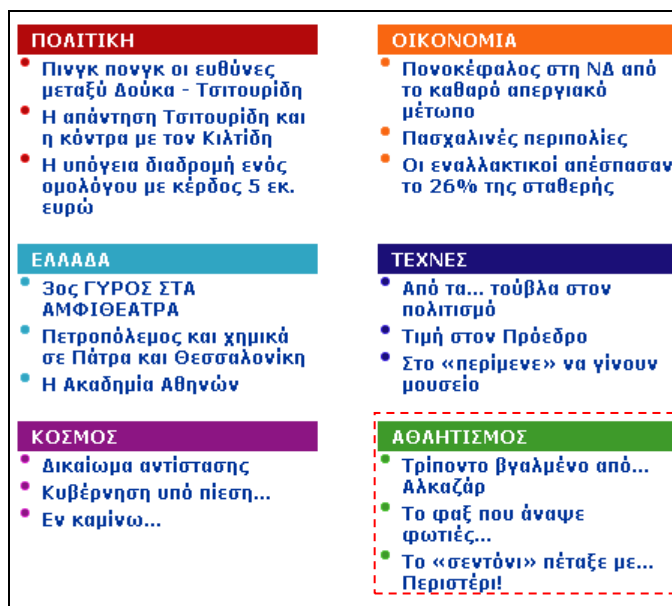


Figure 23: Headers from a sample news website

Obviously, all the headers share the same HTML structure which is shown in Figure 24, which encumbers the isolation and extraction of the desired information.

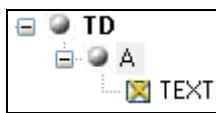


Figure 24: News header structure

To deal with this problem, there is a need to specify the environment of the pattern root. Using the features discussed above, the user can easily build the pattern shown in Figure 25. This pattern returns only the desired pieces of data. It should be noted that the pattern uses the name of the wanted news category as a landmark, ‘ΑΘΛΗΤΙΣΜΟΣ’ for our example, which is described via a regular expression.

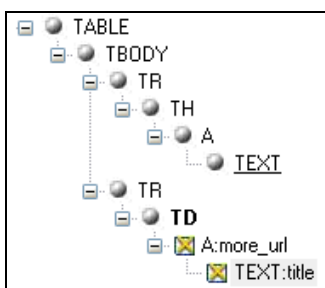


Figure 25: Pattern for sports news headers

As shown in Figure 25, the root of the pattern is the TABLE node while the initial root was the TD node which is in bold. The subtree rooted at TD represents a record instance and hence this is the pattern searched in myDOM. The nodes above

TD are environment constraints. While execution, when a myDOM subtree matches with the subtree rooted at TD , then it is checked for its neighborhood. Only if all constraints are fulfilled, there is a hit.

We call this technique *virtual root* method, as the root of the tree searched in myDOM is not the real pattern root node but the root of the subtree corresponding to a record instance.

Successive Optional Nodes

While pattern matching, once the algorithm cannot match an optional node, it continues with its next sibling, if this exists. However, in a few cases, some successive optional nodes go always together because they belong to a group. So, it is useful to be able to handle multiple pattern nodes as a group. This means that in case an optional node is not found, a certain number of following successive nodes has to be skipped. This is achieved via the *FSON* (Following Successive Optional Nodes) parameter contained in an optional node. The user can assign a value to it through the node popup menu.

Consider the record in Figure 26. Suppose that the director and actors HTML segments are optional in the contrary of the movie title which is required in a record instance.



Figure 26: Record with optional data fields

In the pattern a user would build, he should pose the nodes corresponding to director and actors as optional as in Figure 27. The first `TEXT` node contains the regular expression ‘Director’ while the second has the regular expression ‘Actors’.

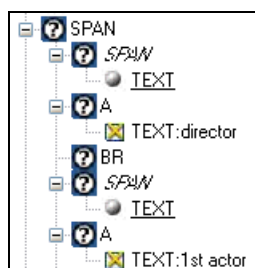


Figure 27: Rule subtree with successive optional nodes

However, in an instance without given director, the first `SPAN` would fail to match but the following `A` node would match with the `myDOM` node `A` of the first actor, which is wrong since the specific pattern `A` node goes with the nodes, `SPAN` and `BR`, belonging to the director.

Let's assume that the user gives the `FSON` of the director `SPAN` the value of 2. Now, if `SPAN` is not matched, the algorithm will jump to the `SPAN` of actors, since it skips the two following successive nodes (`A` and `BR`). As a result, the problem described above is dealt with.

Pattern Matching Algorithm

The algorithm used for pattern matching is really effective in most cases. An extraction rule describes the structure a record instance should have. However, some times this is not enough due to the often occurrence of the specified structure in the target document (e.g. we want the sport headlines on a news site, but all headlines have the same structure). The mechanism we deployed to address this problem is to take into account the neighbourhood / ancestors of the root node of the record instance. Therefore, a ΔEiXTo generated extraction rule can have a virtual root and consequently is made up of two parts. Most times though, the virtual root coincides with the real root. Let `R` be the tree extraction rule and `vroot` the virtual root of `R`. Let `T1` be the subtree of `R` rooted at the `vroot` node, while `T2` is `R-T1`, which consists of the nodes above `vroot`. `T2` is the neighborhood of `vroot`. In case the `vroot` and the real root are the same node, then `T2` is empty.

To identify instances of information under interest, we examine every node of the `myDOM` data structure of the fetched page. To be more specific, at each cycle we try to match the pattern over the subtree of the node under examination. For every single `myDOM` node, a new pattern matching effort begins. Every node in a tree can be seen as the root node of the subtree rooted at that node. So, let `node` be the `myDOM` node examined and `S` the subtree rooted at this node. The algorithm consists of two main steps. In the first step we check if `S` matches `T1` and in the second the neighborhood / environment of `node` is checked for match with `T2`. If both checks are successful, all constraints are fulfilled then there is a hit, which means that a record instance has been detected, so its data is extracted.

The basic idea behind the algorithm is that in order to match two nodes, they must have the same tag and their children must match as well. So, the pattern matching problem becomes a depth first recursive problem. Main attributes of the algorithm is the support of missing nodes in the target tree and the existence of optional nodes in the pattern. The procedure of matching a node of S with a node P of T_1 is based upon first occurrence. Thus, in a cycle the algorithm parses the nodes of the level of S and stops the search of a match for P , when it finds the first node of S that matches P . It should be noted that the pattern matching for a node continues from where the last match occurred.

Consequently, when a myDOM node matches with a pattern node, that means that their children (and recursively their whole subtrees) have also matched already. When a match occurs, then the content of the pattern node fills with the data of the corresponding myDOM node. Thus, in case the whole pattern tree matches, then all the nodes (except optional (and their subtrees) perhaps) have obtained data, some of which are those under interest and so they are extracted.

In the case a required pattern node does not match, then the procedure fails and a new pattern matching cycle begins with the next myDOM node. If a match is not found for an optional node, then its subtree remains empty of data and the algorithm continues with its next sibling node, if there is one. The matching effort continues from the node for which the last match occurred. In the case the optional group handling is enabled then the algorithm continues with the node following the optional group, if there is one.

If the pattern tree matches, which means that a record instance is found, then the extraction of the specified fields of data is performed and the pattern empties again so as to begin a new pattern matching effort with the next myDOM node.

These above described will be better understood with an example. The next figure demonstrates the pattern tree on the left and a sample target myDOM subtree on the right. We assume that all nodes of the pattern are required besides \circ , which is optional and its optional group size is 1 (just itself). The virtual root of the pattern is X and the pieces of data we want to extract are held by K , N and F . Assume that the myDOM element under examination is X . The X nodes have the same tag name but in order to match, their children should first match. In our case, this is done although there is no \circ in the myDOM subtree, since it is optional. Note that T matches with the

second \mathbb{T} myDOM node because the first instance of a \mathbb{T} node has no children. Obviously, all the required pattern nodes that are \mathbb{X} descendants match with the corresponding myDOM nodes. Given that \mathbb{X} nodes match, then their ancestor nodes above them should be matched. Finally, all pattern nodes match, whereupon there is a hit. So, items of data contained in \mathbb{K} , \mathbb{N} and \mathbb{F} nodes are extracted. After that happens, a new tree matching effort will start with another myDOM node examined and so the procedure keeps on.

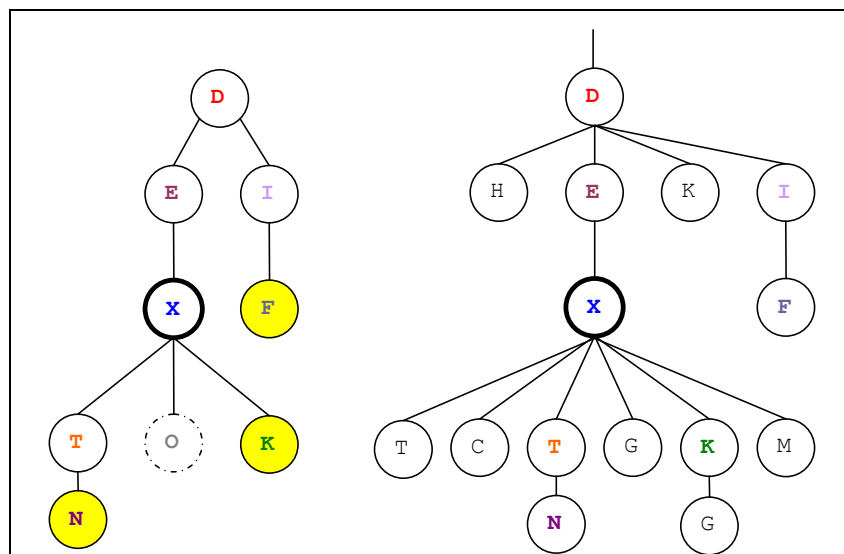


Figure 28: Example pattern and target tree

Auto Mode

Once the user builds an extraction rule that has efficient performance and extracts the desired data, he can save it for future use and execute it at will. Therefore, he does not have to create it over and over again from scratch for the same pages under interest.

All the necessary wrapper information is stored in an XML encoded file, so that the user can load it and run it. These files are named *wrapper project files* and have *wpf* extension, while they follow the syntax rules that the DTD (*wpf.dtd*) poses for their validity check. To open and save *wpf* files, there are relevant buttons (Figure 29) on the *Project Info* tab. It should be noted that in order to open a *wpf*, there must be the *wpf.dtd* in the same directory with the *wpf*.

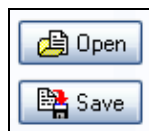


Figure 29: Buttons to open and save a wrapper

A wrapper can be executed for multiple URLs. This is meaningful for pages of the same structure and type, for example pages of the same website. The wrapper visits each one of them, applies the pattern to identify record instances, gathers records and presents the results unified in a single result set. For this purpose, the user can define as input either a list of URLs or a text file containing target URLs.

The specification of the target pages is achieved via relevant controls (Figure 30) on the *Project Info* tab. Note that when a user visits a page, then its address is automatically inserted in the list at the region 1 of the Figure 30, discarding its previous contents first.

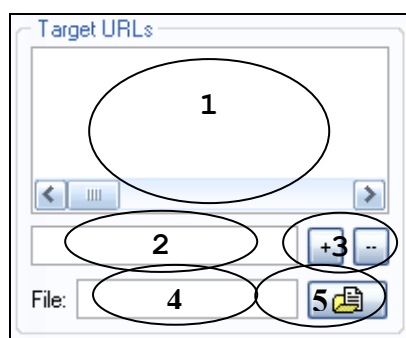


Figure 30: Defining target URLs of a wrapper

The user, through the ‘+’ button and the relevant text field, shown in regions 3 and 2 respectively, can add URLs. Moreover, by selecting a URL of the list and pressing ‘-’, the user can remove URLs. The button on region 5 opens a dialog box that allows the user to open a file and insert the URLs that it contains in the list. The text field at region 4 of the image represents the path of the specified file. The user can enter directly in this text field the absolute or relative path of the file. It should be noted that in this case, no URL insertion is done. Thus, this is useful only when saving a wrapper. Moreover, when the user wishes to save a project, he should select just one of the ways of specifying the target URLs (list or file).

To execute a saved wrapper, the user should load the *project* file via the relevant button (*Open*) and press the button *Go!* (Figure 31). In case the user wishes to stop the execution, he can press the button in Figure 15 in region 2 of the application.

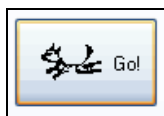


Figure 31: Button for execution of a loaded wrapper

When a wpf is loaded, then the relevant control elements get those values specified in the corresponding XML elements of the project file. The pattern is built and the target addresses are inserted in the relevant list. Double click on an address contained in the list opens the specific page in a new Internet Explorer window. In case some types of nodes should be ignored, then tag filtering procedure is enabled and the relevant checkboxes are checked. Moreover, the controls for output file get the appropriate values, as well as the controls for following ‘Next’ links.

Special interest has the capability to define multiple targets via a file because it makes possible to combine different wrappers and to use the output of one as input to the other. For example, suppose that a wrapper (w1) extracts from one or more pages of a web site the URLs to which the really desired information is located (e.g. product detail pages) and stores them in a text file. A second wrapper (w2) can use as targets the URLs w1 extracted to a file. So, w2 visits all these pages and extracts the actual data under interest. This way, a kind of wrapper cooperation is supported, which is quite important.

It should be noted that DEiXTo can also be executed from command line with parameter the wpf file that contains all the necessary information for the specific run. So, it is possible to set wrappers to run automatically by making use of a job scheduler, such as *Scheduled Tasks* in Windows XP.

Auto Fill and Submit Form

DEiXTo provides the ability (in auto mode) to automatically fill a form, submit it and execute a wrapper on search result pages. Specifically, the procedure is: the search field is filled in with the user term, it is submitted, the first result page is fetched, the record instances are extracted and the procedure goes on with next pages, since the wrapper can follow ‘Next’ links.

This is really helpful for data extraction from pages of search engines, e-shops and price comparison engines. The user should define the form name, the search field name and his search term(s). The two first are *optional*. If the user enters only the

search term, then the *first* form element is selected and its *first* field is filled. This information is provided by the user on *Project Info*, as shown in Figure 32.



Figure 32: Controls for auto fill and submit form

So, the user is not forced to provide URLs for certain search terms as input to the wrapper, e.g. the first result page for the x product type. He can enter manually the website home page as target URL and fill the relevant fields described previously. Each time he wants to extract data from this website, independently of the query, he should execute the same wrapper changing only the search keyword.

Extraction Rule Maintenance

Once the user builds a rule, he can then save it and run it at will. However, due to layout changes of the target web site, a wrapper could stop working as expected. It is also possible that the user needs have changed or the user wishes to modify the pattern for some reason. So, there is a maintenance issue. Of course, the goal is to be able to easily modify and fine tune the pattern so as the user does not have to build a new rule from scratch. Thankfully, most sites are not doing huge revamps often.

The pattern has no data and the tree structure in *Project Info* is read-only. What we want is to find a record instance that matches with the pattern and put it in the record instance tree component, so that the user can edit it and adapt it according to the new needs. This is achieved via the tuning feature.

When user presses the `Tune` button (Figure 33) on *Project Info* tab, then the browser retrieves the target URL and searches in the myDOM tree for a full match with the pattern, so that a record instance is created, which will have data in all nodes.



Figure 33: Tune button

The procedure of finding a full match stops either when this is found or when all target URLs are visited and there is no match. In the record instance identified, the

user can make the necessary changes and configurations to improve the precision and the efficiency of this wrapper. In case, the layout of a site has changed largely, there is a possibility not to be able to find a match. If a match is not found, then no rule-instance is produced and a relevant message is displayed.

Output to RSS File

DEiXTo can produce RSS files for those sites that have their own RSS feeds. The *item* elements of *channel* are created from data extracted from record instances identified. In the *RSS Channel* tab in region 5 of the application the user can define the values of the sub-elements of the *channel* element of the RSS output file (Figure 34). The sub-element *title* gets automatically the value “ΔEiXTo: page title”, except the user enters his own title.

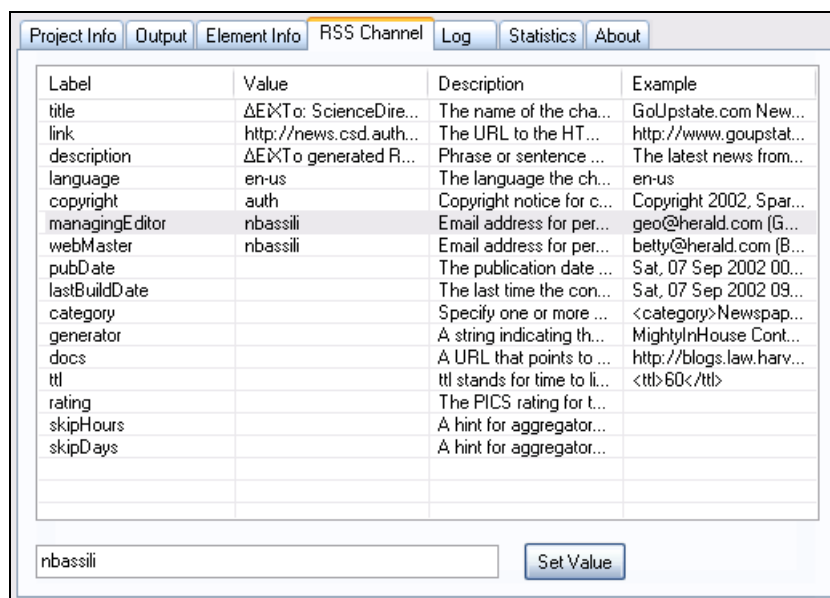


Figure 34: Sub-elements of channel element of RSS output file

The user can assign an RSS label to each node of the pattern, as shown in Figure 35. He can choose between RSS elements: *title*, *author*, *description*, *link* and *pubDate*. In case the user has not assigned a ‘link’ label to a node of those extracted, then automatically a link element is added in each ‘item’ element, which has the value of the address of the page from which the record derived. To execute an extraction rule generating a RSS feed, it is required that the pattern contains a node that has a *checked* or *checked_implied* state and has a *RssTitle* or *RssDescription* label.

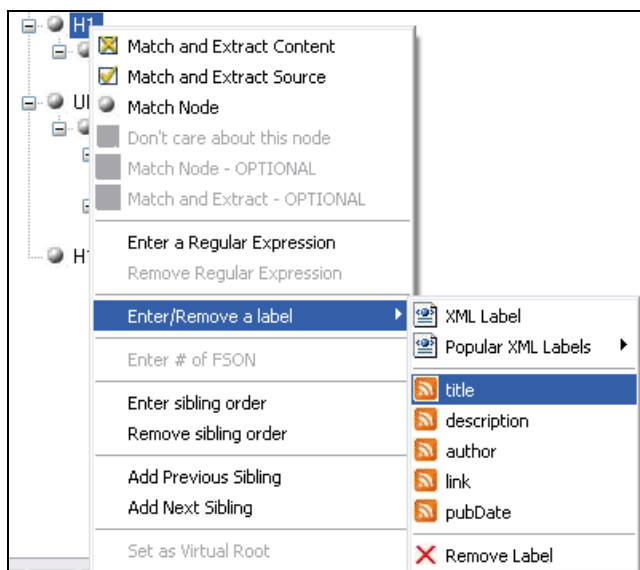


Figure 35: Assignment of RSS label to a rule node

Sibling Order

Until now, according to the algorithm description, pattern matching is based on the order of node occurrence but not on sibling order. However, in some cases, it is useful that the user can define the sibling order of a node. This is done through a relevant option from the local node menu. In the window that opens (Figure 36), the user can provide mathematical expressions of type $\kappa * i + C$, where C is the start index, κ is the step and i is an integer greater than or equal to (\geq) 0.

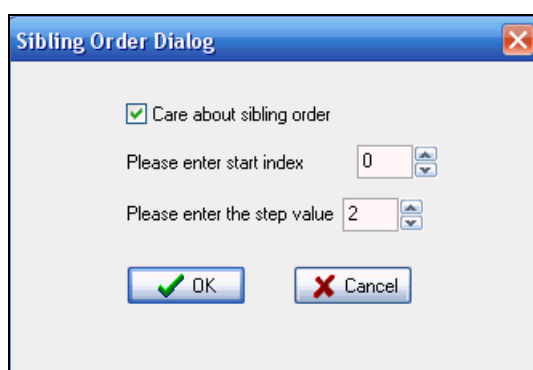


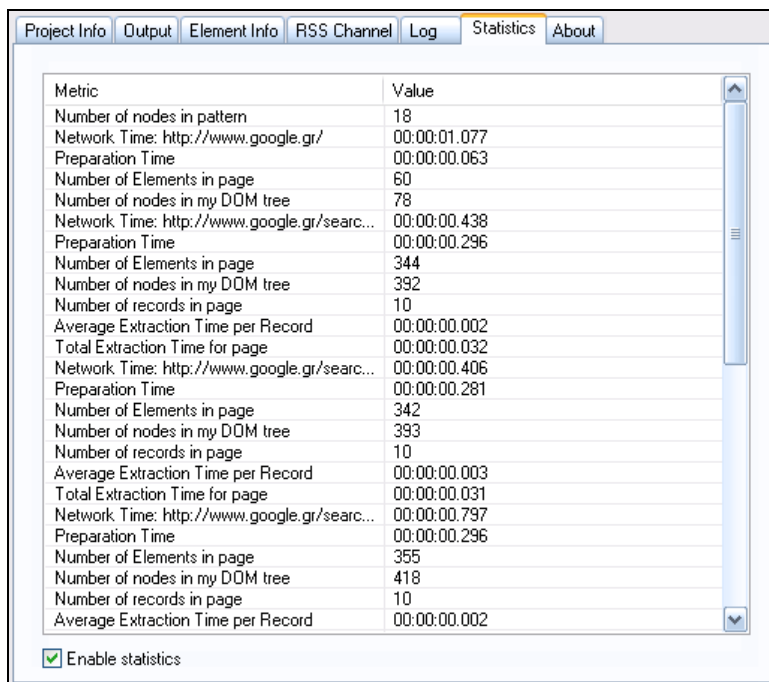
Figure 36: Dialog box for sibling order definition

It should be noted that the first child’s sibling order is 0. If the user wants a constant sibling order N , he can then give value 0 to step and N to start. So, if the user would like a pattern node to match with myDOM nodes with even sibling order, 0,2,4,6,..., he should give to the start index the value 0 and to step the value 2. A sample example of this function would be the extraction of odd or even search results.

Statistics

In *Statistics* tab in region 5 of the application windows, there are provided some metrics concerning the execution time and the system performance (Figure 37). Specifically, if the statistics checkbox on the same tab is checked, then the metrics measured are:

- Network time: time needed to fetch a page and fully render it in the browser.
- Preparation time: time needed to build the myDOM tree of a page and the necessary data structures.
- Number of HTML elements on a page.
- Number of myDOM nodes of a page.
- Number of nodes in an extraction rule.
- Number of record instances identified in a page.
- Total extraction time for the desired data of a page.
- Average extraction time per record.



Metric	Value
Number of nodes in pattern	18
Network Time: http://www.google.gr/	00:00:01.077
Preparation Time	00:00:00.063
Number of Elements in page	60
Number of nodes in my DOM tree	78
Network Time: http://www.google.gr/search...	00:00:00.438
Preparation Time	00:00:00.296
Number of Elements in page	344
Number of nodes in my DOM tree	392
Number of records in page	10
Average Extraction Time per Record	00:00:00.002
Total Extraction Time for page	00:00:00.032
Network Time: http://www.google.gr/search...	00:00:00.406
Preparation Time	00:00:00.281
Number of Elements in page	342
Number of nodes in my DOM tree	393
Number of records in page	10
Average Extraction Time per Record	00:00:00.003
Total Extraction Time for page	00:00:00.031
Network Time: http://www.google.gr/search...	00:00:00.797
Preparation Time	00:00:00.296
Number of Elements in page	355
Number of nodes in my DOM tree	418
Number of records in page	10
Average Extraction Time per Record	00:00:00.002

Enable statistics

Figure 37: Statistics for execution of a sample wrapper