Defining a new language

Introduction

Defining a new language is necessary if you want to use the Universal Analyzer to directly analyze source code, or you want to use the **Universal Importer** tool to import data generated by an external analyzer. The first step involves defining the language and then the objects that form the language you want to analyze. This is done by editing the **xxxMetaModel.xml** file in your package.

Please note the following rules pertaining to the naming of types and categories in your xxxMetaModel.xml file:

- you cannot declare a type/category name starting with CAST_
- a type/category name can only contain letters [A-Z], numbers [0-9] and the "_" character

Language category

Universal Analyzer

To add a new language specifically for the **Universal Analyzer**, you must first define a new category for the new language. This category must contain:

- The language category name and rid: <**category name="language-name" rid="integer">**
- The language description: **<description>language-description</description>**
- The file extensions: **<attribute name="extensions" stringValue="ext1[;ext2;...]"/>** - if there are several extensions, use a semicolon ";" to separate them (i.e.: "**.jsp;.jspx**").

This category must also inherit from the predefined categories below:

- The "**UniversalLanguage**" category (so that this category will denote a language handled by the Universal Analyzer)
- The "**CsvLanguage**" category

Take the following example:

```
<category name="PHP" rid="0">
   <description>PHP</description>
   <attribute name="extensions" stringValue="*.php;*.php3;*.php4;*.php5;*.php6;*.inc"/>
   <inheritedCategory name="UniversalLanguage"/>
   <inheritedCategory name="CsvLanguage"/>
</category>
```

Universal Importer

To add a new language specifically for the **Universal Importer**, you must first define a new category for the new language. This category must contain:

- The language category name and rid: <**category name="language-name" rid="integer">**
- The language description: **<description>language-description</description>**

Note that **file extensions** do not need to be defined for languages that will be imported via the **Universal Importer** - the "source code" will be in .UAX file format for these languages.

This category must also inherit from the predefined categories below:

- The "**UniversalLanguage**" category
- The "**UniversalImporter**" category (so that this category will denote a language handled by the **Universal Importer**)
- The "**CsvLanguage**" category

Take the following example:

```
<category name="Siebel" rid="2">
        <description>Siebel</description>
        <inheritedCategory name="UniversalLanguage"/>
        <inheritedCategory name="ImporterLanguage"/>
        <inheritedCategory name="CsvLanguage"/>
</category>
```

Language project "type"

Following the addition of the category for the language, you must then define a "type" that corresponds to the projects for this language. This type category must contain:

- The project type name and rid: **<type name="project-name" rid="integer">**
- The project description: **<description>project-description</description>**

This type MUST also inherit from the category previously defined for the new language. Using the previous example for the PHP language, this category will thus be "**PHP**". This type MUST also inherit from the predefined category below:

- The "**UAProject**" category

- Please see **UAObject and UAProject categories** in xxxMetaModel.xml file for more information about the **UAProject** category.

Example for the PHP language (cont.):

```
<type name="phpProject" rid="6">
<description>PHP Project</description>
<inheritedCategory name="UAProject"/>
<inheritedCategory name="PHP"/>
</type>
```

Language object types

The next stage is to define the type of objects for your language (functions, classes, etc). For each type of object in the language, there is an element named "type", as for the language project.

To indicate that an object belongs to another (i.e.: "is the child of"), the "tree" element is used. This element allows you to specify the type of the parent and the category of the relation to this parent. One type of object can have several types of parents. For example, a Java class can have a file and another class as a parent. You must use as many "tree" elements as there are possible types of parents.

A type which represents the file already exists in the metamodel, so there is no need to define a new type for a source file. This type is named **sourceFile**. Thus, each object that has the file as a parent must contain the following information:

```
<tree parent="sourceFile" category="amtParentship"/>
```

This object type MUST also inherit from the predefined category below:

- The "**UAObject**" category

Please see **UAObject and UAProject categories** in xxxMetaModel.xml file for more information about the **UAObject** category.

The following example (for the PHP language), shows how the PHP language functions are defined. Note that their parents are "sourceFile" objects, since PHP source files may contain PHP functions:

```
<type name="phpFunction" rid="5">
   <description>PHP Function</description>
   <inheritedCategory name="UAObject"/>
   <inheritedCategory name="CalleeLinkable"/>
   <inheritedCategory name="CallerLinkable"/>
   <inheritedCategory name="METRICABLE"/>
   <inheritedCategory name="PHP"/>
   <inheritedCategory name="caseInsensitive"/>
   <tree parent="sourceFile" category="amtParentship"/>
   <tree parent="EnlightenPHP" category="EnlightenTree"/>
</type>
```

Dealing with case sensitivity

Universal Analyzer supports case sensitivity for all object types defined by you the user. By default, objects defined in metamodel files *are case sensitive*. Case insensitivity is obtained by inheriting from the "caseInsensitive" category. If an analyzed language is globally case insensitive you may have your language category inherit from "caseInsensitive" instead of each type in the language.

Case sensitivity applies to both object identification and object search. Namely, searches for case insensitive objects are case insensitive. For instance, PHP interfaces, classes, methods and functions are case insensitive, but PHP members are case sensitive:

```
<type name="phpInterface" rid="1">
...
   <inheritedCategory name="caseInsensitive"/>
...
</type>
```

Please remember that all source files (represented by the category **sourceFile**) are always **case sensitive** (i.e. this is true for all Universal Analyzer languages). This is default behavior defined by CAST and cannot be changed by the user.

Escalation process configuration

The escalation process between objects of a language is configured using one or several of the categories listed below. Objects whose type has none of the escalation-related categories will not be part of the escalation process, and none of their direct children will escalate up to them.

There are four basic escalation-related categories:

ESCALATION_AS_CALLER_CHILD	When this object is "the caller" in a link, this object will be escalated up to its parent provided that the parent allows objects to be escalated up to it.
ESCALATION_AS_CALLEE_CHILD	When this object is "the callee" in a link, this object will be escalated up to its parent provided that the parent allows objects to be escalated up to it.
ESCALATION_AS_CALLER_PARENT	When this object is "the caller" in a link, this object allows its direct child objects to be escalated up to it.
ESCALATION_AS_CALLEE_PARENT	When this object is "the callee" in a link, this object allows its direct child objects to be escalated up to it.

Based on these four categories, the following categories are built using inheritance:

ESCALATION_ALL	Inherits from all of the four basic escalation-related categories. Consequently, whether this object is "the caller" or "the callee" in a link, this object will be escalated up to its parent (provided that the parent allows objects to be escalated up to it), and will allow its direct child objects to be escalated up to it.
ESCALATION_AS_CHILD	Inherits from ESCALATION_AS_CALLER_CHILD and from ESCALATION_AS_CALLEE_CHILD. Thus, whether this object is "the caller" or "the callee" in a link, this object will be escalated up to its parent provided that parent allows objects to be escalated up to it.

ESCALATION_AS_PARENT	Inherits from ESCALATION_AS_CALLER_PARENT and from ESCALATION_AS_CALLEE_PARENT. Thus, whether this object is "the caller" or "the callee" in a link, this object allows its direct child objects to be escalated up to it.
ESCALATION_AS_CALLER	Inherits from ESCALATION_AS_CALLER_CHILD and from ESCALATION_AS_CALLER_PARENT. Thus, when this object is "the caller" in a link, this object will be escalated up to its parent (provided that the parent allows objects to be escalated up to it), and will allow its direct child objects to be escalated up to it.
ESCALATION_AS_CALLEE	Inherits from ESCALATION_AS_CALLEE_CHILD and from ESCALATION_AS_CALLEE_PARENT. Thus, when this object is "the callee" in a link, this object will be escalated up to its parent (provided that the parent allows objects to be escalated up to it), and will allow its direct child objects to be escalated up to it.

Configuration of links between objects

An object can have various links to other objects. Because links are directional, you must define if the object is the link source (represented by "CallerLinkable") or the link target (represented by "CalleeLinkable") (or both) for each link which can have an object.

You must use the tree tag as shown below:

```
<tree parent="LinkType" category="CallerLinkable"/>
<tree parent="LinkType" category="CalleeLinkable"/>
```

The link types that are defined in the metamodel file are:

```
accessLink
accessMemberLink
accessOpenLink
accessPageForwardLink
accessPageIncludeLink
accessReadLink
accessWriteLink
----------
callGotoLink
callLink
callPerformLink
callProgLink
callTransacLink
----------
catchLink
----------
containDeclareLink
containDefineLink
containLink
----------
ddlAlterLink
ddlCreateLink
ddlDropLink
ddlLink
ddlReplaceLink
----------
dynamicLink
staticLink
----------
fireAfterLink
fireInsertLink
fireInsteadOfLink
fireLink
fireSelectLink
fireUpdateLink
----------
friendLink
----------
gothroughLink
----------
includeLink
----------
inheritExtendLink
inheritHideLink
inheritImplementLink
inheritLink
inheritOverrideLink
----------
internallyEscalatedLink
----------
```

joinLink
joinNonSysCatalogLink
joinSpCommonKeyLink
joinSpForeignKeyLink
---
lockLink
---
matchLink
---
mentionLink
---
monitorForAllRowsLink
monitorForEachRowLink
monitorInsertLink
monitorInsteadOfLink
monitorLink
monitorSelectLink
monitorUpdateLink
---
prototypeLink
---
raiseLink
---
referCascadeLink
referDeleteLink
referInsertLink
referLink
referSetnullLink
referUpdateLink
---
relyonIsInstanceOfLink
relyonLink
---
throwLink
---
useDeleteLink
useInsertLink
useLink
useSelectLink
useUpdateLink

Example, for the **inheritLink**, a PHP class is **CallerLinkable** (because a class can inherit from another object) and **CalleeLinkable** (because another object can inherit from a class):

```
<type name="phpClass" rid="2">
...
   <tree parent="inheritLink" category="CallerLinkable"/>
   <tree parent="inheritLink" category="CalleeLinkable"/>
...
</type>
```

Note that:

- If an object type inherits directly from the categories "CallerLinkable" [example : **<inheritedCategory name="CallerLinkable"/>**] or is at least caller of **one** link type, the internal and external link search will be activated on this object.
- If an object type inherits directly from the categories "CalleeLinkable" [example : **<inheritedCategory name="CalleeLinkable"/>**] this object is the callee for **all** link types. Using the inheritance of these categories is just a shortcut, but loses precision.

Syntactic definition in language pattern file

Similar to the process of object recognition, regular expressions can be used to define how links are recognized: first the pattern to be recognized is specified and then the identifier of the called object is retrieved.

For example, to recognize an "extend" type link between two classes:

```
<inheritLink>
<pattern><![CDATA[extends([ ]|[\r\n]|[\t])+]]></pattern>
<callee><![CDATA[[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]]]></callee>*
</inheritLink
```

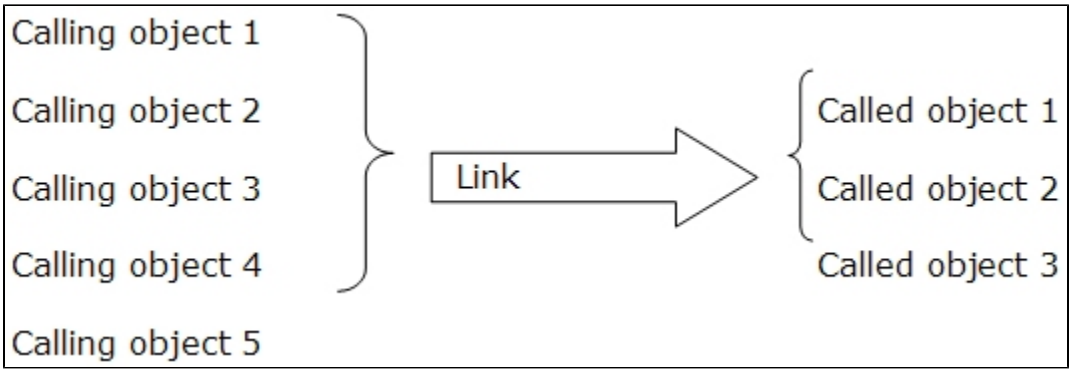
Semantic definition in metamodel

During the object type definition process, the object is defined as calling or called for specific link types (below: a PHP class is both calling and called for an "inherit" type link).

For example, PHP class definition (the list of links is not exhaustive):

```
<type name="phpClass" rid="2">
   <description>PHP Class</description>
   <inheritedCategory name="UAObject"/>
   <inheritedCategory name="CalleeLinkable"/>
   <inheritedCategory name="CallerLinkable"/>
   <inheritedCategory name="PHP"/>
   <inheritedCategory name="APM Classes"/>
   <inheritedCategory name="caseInsensitive"/>
   <inheritedCategory name="AUTO_VIEW_HIGH_LEVEL_OBJECT"/>
   <tree parent="inheritLink" category="CallerLinkable"/>
   <tree parent="inheritLink" category="CalleeLinkable"/>
   <tree parent="sourceFile" category="amtParentship"/>
   <tree parent="EnlightenPHP" category="EnlightenTree"/>
</type>
```

It is therefore very important to understand that the link is in fact the "**search pivot**": in fact each link is specified as either calling or called. As a result, there is no direct linkage between one object and another; however there is a linkage between the calling object and the link and a linkage between the link and the called object:



In order to retain coherence with existing specifications, if the user indicates: **<inheritedCategory name="CallerLinkable"/>** this signifies that the object is called for all link types (this corresponds to adding the tags **<tree parent="…" category="amtCalleeship"**> for each defined link type). If the user indicates **<inheritedCategory name="CallerLinkable"/>** or if the object is at least caller of one link type, the internal and external link research will be actived on this object.

Link search mechanism

Link resolution is carried out as follows during the processing of an object X:

- Object name "blanked out"
- Identification of object's child objects; child objects' source code is "blanked out"

Search for link types for which the object X is defined as a "caller": for each link, the regular expression is searched for – this allows the link to be found (regular expression is defined in **XXXLanguagePattern.xml**) – and then a reference to the object called by the link is created. This reference contains:

- The "caller" (object X)
- The link type
- The code of the callee

E.g.: **class Bicycle extend Vehicle.2wheels** : *caller* = **Bicycle,** *link type* = **extendLink**, *code* = **Vehicle.2wheels**

The links that have been identified are "blanked out" so that they are ignored during the external linkage process. A "Blanked out" body is emitted so that object X is processed as a Universal caller through the external linkage action (if the object is a caller of a link type, or if it inherits from the "CallerLinkable" category"). In addition, if the search for "non-typed" intra-UA links is activated (TRUE by default), the "blanked out" body is emitted so that it is processed via the internal linkage action (if the object is a caller of a link type, or if it inherits from the "CallerLinkable" category"). After having examined all the objects, the references that have been created are then processed. For each reference:

- Search for potential "Callee" objects for the reference's link type
- Search for these objects in the link code:

- If any are found, a link is created between the reference's caller and the object that has been found – this link is defined in the reference.
- If none are found, an external reference is created so that the link is recognized by the external linkage action.
- If the search for intra-UA "non-typed" links is activated, a search is made in the "blanked out" body of the UA objects for any references to other UA objects (if they are a callee of a link type). A USE link type is created whenever a reference is found.

Finally, a search is carried out for links via the external link action (i.e.: on all the objects present and selected in the analyzer object browser). This is valid for:

- The "typed" links that were not resolved via the internal linkage action (these lose there type filter in this case).
- The "blanked out" bodies of all objects (if they are "caller").

Binding icons and pictures to object types

You can bind your own icon and/or picture to each object type in your newly defined language so that CAST Enlighten and the CAST Discovery Portal can use the correct images to represent the object types. By default, .ICO files are used in CAST Enlighten and .PNG files are used in the CAST Discovery Portal. .WMF files can also be used in CAST Enlighten, but only in addition to the .ICO files.

If .ICO and .WMF files have been added or modified you must validate and re-save your package using the UA Assistant (***UAAssisstant.exe***), however no reinstallation in the CAST Analysis Service is necessary (using CAST Server Manager). CAST Enlighten must also be closed and re-launched for .ICO and .WMF files to be taken into account.

Configuring .ICO files for use in CAST Enlighten

For each object type for which you want to display an icon:

- Create a single .ICO file containing two icons representing objects of that type (first icon must be 16x16 pixels and the second icon must be 32x32 pixels) – note that .ICO files can contain more than one icon.
- Name the single .ICO file with the same name as the object type it represents in the metamodel, e.g. for PHP functions: **phpFunction.ico** for **<type name="phpFunction" rid="5">**
- Save the .ICO file in the "**Res**" subfolder of the language package directory.
- CAST Enlighten will then automatically use either of the icons in the .ICO file depending on the situation.

Configuring .WMF files for use in CAST Enlighten

- By default, CAST Enlighten will use the corresponding icon file - .ICO - (if none are found, a question mark will be displayed instead). This may, however, produce views that are not aesthetically pleasing if the icon's zoom factor becomes too large. In this case, you can also:
- Create a .WMF file containing a picture representing your object type (recommended size is 50x50 pixels)
- Name that .WMF file with the same name as the object type it represents, e.g. for PHP functions: **phpFunction.wmf** for **<type name="phpFunction" rid="5">**
- Save the .WMF file in the "**Res**" subfolder of the language package directory.
- CAST Enlighten will then automatically use the WMF file instead of the .ICO file depending on the situation

Configuring .PNG files for use in the CAST Discovery Portal

If you are planning to use the CAST Discovery Portal with your newly defined language, you MUST add .PNG files (Portable Network Graphics files) to a separate location:

- Create a PNG file containing a picture representing your object type (recommended size is 16x16 pixels)
- Name that .PNG file with the same name as the object type it represents, e.g. for PHP functions: **phpFunction.png** for **<type name="phpFunction" rid="5">**
- Save the .PNG file in the "**themes/default/objects**" subfolder at the location used to store your CAST Discovery Portal web application. Typically (in a Windows/Apache Tomcat environment) this will be: **%CATALINA_HOME%\webapps\CASTAD**
- The CAST Discovery Portal will then automatically use the PNG file to represent your object. If no corresponding .PNG file can be found, a missing image icon will be displayed instead (typically a blank square with a red cross inside).

Embedded SQL support

If the language you want to analyze includes embedded SQL (i.e. calls to server-side database objects), then you need to tell the Universal Analyzer which objects need to be searched for embedded SQL. This can be done by ensuring that each object that needs to be searched for embedded SQL inherits from the **ESQLSearchable** category.

For example:

```
<inheritedCategory name="ESQLSearchable"/>
```

So for a PHP class:

```
<type name="phpClass" rid="2">
   <description>PHP Class</description>
   <inheritedCategory name="UAObject"/>
   <inheritedCategory name="CalleeLinkable"/>
   <inheritedCategory name="CallerLinkable"/>
   <inheritedCategory name="PHP"/>
   <inheritedCategory name="APM Classes"/>
   <inheritedCategory name="caseInsensitive"/>
   <inheritedCategory name="AUTO_VIEW_HIGH_LEVEL_OBJECT"/>
   <inheritedCategory name="ESQLSearchable"/>
   <tree parent="inheritLink" category="CallerLinkable"/>
   <tree parent="inheritLink" category="CalleeLinkable"/>
   <tree parent="sourceFile" category="amtParentship"/>
   <tree parent="EnlightenPHP" category="EnlightenTree"/>
</type>
```

Please make sure you also modify the corresponding **xxxLanguagePattern.xml** file in your language package so that the beginning and end of each embedded SQL call is defined in the **<esql>** tag. You can find out more about this in the section **Embedded SQL support** in xxxLanguagePattern.xml - defining how to analyze a language.

Tips for embedded SQL support

Some of the more commonly asked questions with regard to configuring embedded SQL support are listed below:

- What can be expected when an object is set to inherit from the **ESQLSearchable** category and an embedded SQL query is identified?
  - the embedded SQL query will be parsed and appropriate links will then be created
  - the Metrics Assistant will be run to calculate any relevant metrics
- Are "embedded SQL query" objects created and saved to the Analysis Service?
  - No
- When an object inherits from the **ESQLSearchable** category and when a link is identified, what link types can be created?
  - **Us** (Use Select), **Ui** (use Insert), **Uu** (Use Update), **Ud** (Use Delete) type links can be created
- Are "All Technology" type Quality Rules calculated by default if an object inherits from the **ESQLSearchable** category?
  - Yes
- Are "SQL metrics" (i.e. Number of FROM, number of WHERE clauses) calculated?
  - Yes
- Are Quality Rules that check the SQL metrics calculated automatically?
  - Yes