Elsevier Fingerprint Engine™

Documentation

Date: April 2015

Elsevier Research Intelligence

CONTENTS

CHAPTER

ONE

ELSEVIER FINGERPRINT ENGINE

The Elsevier Fingerprint Engine is a text processing framework which can be extended with custom components. The aim of this document is

- to give an overview of the Elsevier Fingerprint Engine,
- to get a power-user started with installing and configuring the Elsevier Fingerprint Engine and
- to give an application programmer an idea of how the Elsevier Fingerprint Engine can be called from a client application.

The Elsevier Fingerprint Engine that performs text analysis has been code-named as ‘TACO’ in development. In the programming interface you will find the acronym TACO (Text Analysis COmponents) as coded namespace for the Elsevier Fingerprint Engine.

CHAPTER

TWO

CONCEPTS OF THE ELSEVIER FINGERPRINT ENGINE

2.1 General overview

The Elsevier Fingerprint Engine is a text processing framework, providing the infrastructure and functionality to analyze and annotate text. In principle the Fingerprint Engine framework does not limit the type of analysis that can be performed - it only provides the infrastructure for doing that. To this end, its structure is very generic, allowing for various types of text processing. A configured instance consists of a number of elements:

- Activities
- Annotations and annotation filters
- Workflows
- Static objects
- Preprocessors

These elements will be described in more detail below. Configuration of a Fingerprint Engine instance is a matter of defining one or more workflows for this instance. Once this is done and the instance is running, client applications can call the Fingerprint Engine to apply a certain workflow to an input text. The result will be an annotated text in XML format as defined by the annotations in the corresponding workflow.

2.2 Activities

Activities are text processing steps performing one particular task. An activity is, e.g., sentence boundary detection breaking up a text into separate sentences. A series of different activities then forms a more complex text processing task. For instance, combining the activities *language detection*, *sentence boundary detection*, and *normalization*. The order is important, since it is likely that certain activities need as input the output of other activities. An example is part of speech tagging, which needs the output of a tokenizer. *Configuration of an activity* also defines a tag for the output (annotation) of the activity. There is a chapter on *currently available activities*.

2.3 Annotations and annotation filters

Annotations are the output of activities. An activity may need as input one or more annotations, and can produce multiple annotations. Annotations are classified in various *annotation types*. Since workflows involve normally multiple activities, a workflow could produce a large amount of annotations. In *workflow configurations* one can define *annotation filters* that specify which annotations must be produced as output of the entire workflow. Other annotations are then suppressed.

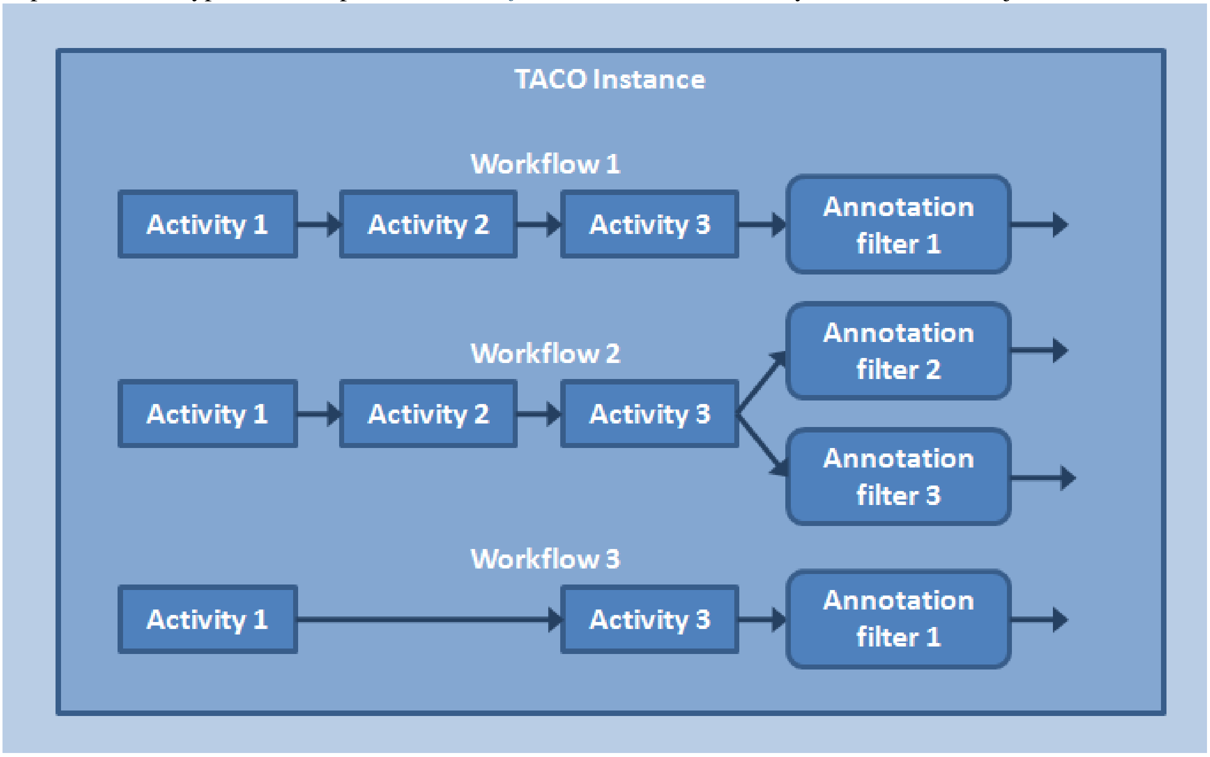
There are several kinds of annotation filters, e.g., whitelist annotation filters that list the items to be included in the output and blacklist annotation filters that define items to be excluded in the output. Typically activities require annotations of other type of activities, e.g., the part of speech tagger activity needs a tokenizer annotation. This dependence is specified in the *workflow configuration* by referring in an activity to the to be used activity annotation tag.

2.4 Workflows

Workflows define which activities should be performed and in what order. Their *configuration* also involves defining one or more annotation filters to define which activity output should be produced as workflow output. Essentially workflows can be regarded as definitions of complex activities.

2.5 Static objects

Static objects are datasets or functions that are available to be used for any activity. Examples of static objects are term list (either as a text file or as a table in a database), a thesaurus, a tokenizer, etc. Certain activities require the presence of static objects of a specific type. The dehyphenation activity, for instance, uses a list of terms to decide whether or not to dehyphenate. Its configuration therefore needs to specify a static object that provides such a term list. Note that static object types are hierarchical, meaning that certain object types inherently also implement other types. The chapter on *static objects* discusses the currently available static objects in detail.

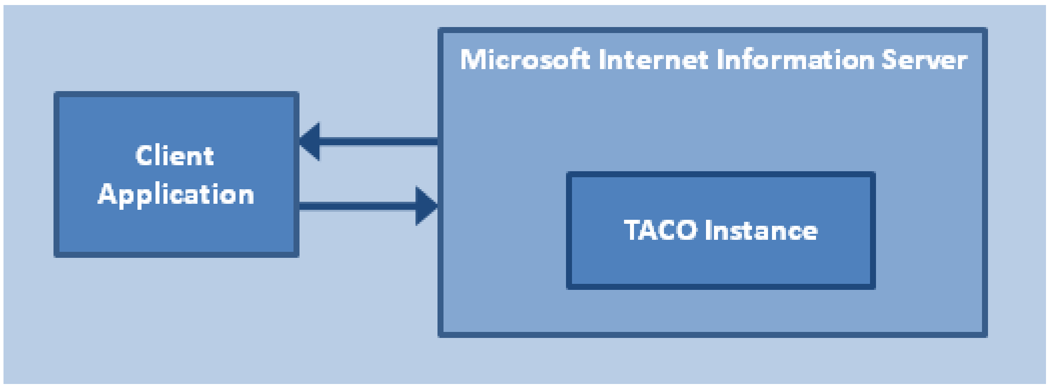


2.6 Architecture

The infrastructure that the Fingerprint Engine provides consists of a number of elements:

- A .Net platform containing a data structure and toolkit to process a text and its analysis;
- A collection of activities for a number of analytical tasks for several languages;
- **A host process**
  - for initializing and running Windows Workflow Foundation workflows defining text analysis functionality;
  - for keeping large and complex data objects for re-use in textual analysis, such as thesauri, dictionaries and normalization tables;
  - A server process that receives text analysis requests, dispatches analysis workflows and communicates the results to the client.

The host process is installed as a web application in Microsoft Internet Information Server (IIS). A typical client application communicates with IIS which in turn dispatches the request to the Fingerprint Engine. The result, annotated text, is returned to the application via IIS. This process is depicted in the figure below.



2.7 Preprocessors

Preprocessors are not so much a functional part of the Elsevier Fingerprint Engine: rather, preprocessors filter or adapt the input in order to deliver the correct input to the Fingerprint Engine, that is: text. *Preprocessors* are separate modules that convert and or format a text to make it suitable to submit to the Fingerprint Engine. For instance, a preprocessor could take an HTML document, extract the text to be processed, and add a section annotation to indicate the title, body etc. The *Tokenize activity* for instance, can accept a section annotation and process it.

CHAPTER

THREE

INSTALLING THE ELSEVIER FINGERPRINT ENGINE

3.1 Installers and deliverables

The following installers and deliverables are available for the Elsevier Fingerprint Engine.

- *EFEDocumentation.exe* This installer installs the documentation (in HTML and PDF format) for installing, configuring and using the Elsevier Fingerprint Engine. It also contains sample code in C# showing how to call the Elsevier Fingerprint Engine.
- *EFEInstallerIIS.exe* This installs a Elsevier Fingerprint Engine instance with a number of workflows.
- *EFEThesaurus*.exe* More or less apart from the core Fingerprint Engine installation, installers may be delivered that copy thesauri backups. For instance, a MeSH thesaurus can be delivered together with MeSH-related workflow configurations. Usually, the installed files are in MS SQL Server backup format that need to be restored into SQL Server using, for instance, the SQL Server Management Console.

Multiple instances of the Fingerprint Engine can be installed next to one another by running the EFEInstallerIIS.exe multiple times. Note then that you do need to assign the various instances different names. This is a part of the install process, where the default name is *taco*.

Installing multiple instances is recommended if you want to be able to load / unload Fingerprint Engine workflows separately. In a scenario where you work with multiple indexing processes, e.g. some using thesauri, some using POS Tagging, it is recommended to install multiple instances. These instances are best distributed over separate IIS Application Pools, which gives full control over the applications. When you use larger thesauri, starting a Elsevier Fingerprint Engine instance may take more time; restarting large combinations of workflows then becomes unwieldy.

Installing the Elsevier Fingerprint Engine requires a number of additional steps to be taken before and after running the installation software,

3.2 Prerequisites

An installation of Elsevier Fingerprint Engine needs a system meeting the following requirements:

- Windows Server 2003, XP, Vista or Server 2008. It is required that your particular flavor of Windows system includes / supports IIS.
- Internet Information Services (IIS) version 6 or 7. The IIS installation is part of the Windows (Server) Operating system and can be configured using the Control Panel, Programs / Software, Windows Components entry point. The Elsevier Fingerprint Engine needs an IIS installation with .Net scripting support.
- .Net support. The Elsevier Fingerprint Engine requires the .Net framework version 4.0. See Microsoft .Net. The installer checks which version of .Net is installed.
- An SQL server 2008 installation (or newer) to contain thesaurus data. For security configuration purposes this is preferably locally installed but a remote SQL Server installation is supported as well. Also SQL Server express is sufficient, but note that it supports only small datasets. For thesauri, however, the limit on database size is high enough.

- Disk space. The binaries for a Elsevier Fingerprint Engine instance take less than 20MB, but obviously more is better. Thesauri sizes range from just a few MBs to 800MB. For good system performance, go with a reasonable minimum of - for instance - 250GB.
- Sufficient RAM space. The bare minimum is 1GB. For good performance, 16GB is recommended. If large thesauri are used, more RAM will improve performance.
- Processor capacity. The Elsevier Fingerprint Engine will work just fine on a single-processor machine. However, the program has been designed with multi-threading in mind, so if you have multiple cores, the Elsevier Fingerprint Engine will use them.

3.3 Steps before installing the software

3.4 Installing the software

Multiple instances of the Fingerprint Engine can be installed next to one another on a single machine. This entails running the TACOInstallerIIS.exe multiple times. Do note that then unique instance names should be supplied during installation. Using a names that was previously using during an installation will overwrite that instance. This is allowed, but of course will result in losing the previous instance. It is recommended, however, to uninstall an instance *with its own un-install program* before reusing the instance name.

The Elsevier Fingerprint Engine installer administers all instances that are installed in the registry.

A Elsevier Fingerprint Engine instance is installed as an IIS Application. The installer suggests the IIS wwwroot-folder as the proposed installation location. However, the Elsevier Fingerprint Engine can be installed on another location. In that case the installer will create a virtual directory in IIS.

After selecting the root folder for installation, an instance name must be specified. The previously installed instance names will be listed. Again note that picking a name that already exists will overwrite that Elsevier Fingerprint Engine instance.

The installation procedure installs a program *UninstallTACOInstance* in the Elsevier Fingerprint Engine root folder. This program should be used to uninstall.

3.5 Steps after installing the software

The TACOThesauri.exe installer will install the thesaurus databases in the SQL Server backup format. To enable SQL Server to use these files, the backups need to be restored. This is best done using the SQL Server Management Console. Note that the backup also contains the definition of the logins and users that are allowed to access the database. After restoring the thesaurus databases one should make sure that access to the database is allowed by the user referred to in the application pool identity.

Once these steps have been taken next steps are *configuring* the installed Elsevier Fingerprint Engine instance and finally *testing* it. There are two files that need configuration: `installation folder\web.config` and `installation folder\config\instance config folder`. The instance config folder is indicated by the web,config file. The details of configuring a Elsevier Fingerprint Engine instance are found in the section on *instance configuration*.

3.6 Starting a Elsevier Fingerprint Engine Instance

The Elsevier Fingerprint Engine installs as an IIS-managed web service. This means that the Elsevier Fingerprint Engine service is started and stopped as an IIS server process. See the section on *instance configuration* how to properly configure an instance. Editing or even just touching the `web.config` file will restart the Elsevier Fingerprint Engine service. Another option is to recycle the application pool the instance is located in. See, e.g., http://technet.microsoft.com/en-us/library/cc735247%28WS.10%29.aspx.

A Elsevier Fingerprint Engine instance consists of one or more workflows that each consist of a number of processing steps. Several of these activities, such as Normalization, Thesaurus-based concept finding or Part-Of-Speech tagging, require a large amount of data in order to perform their tasks. For performance reasons, the data for these activities is stored statically, which means that the data remains loaded throughout the lifetime of the Elsevier Fingerprint Engine instance. However, the IIS server may unload a program if that program remains inactive for a longer period of time. The effect of this is that an initial call to the Fingerprint Engine may take very long, up to a minute, while subsequent calls are handled very fast - as they should be.

3.7 Troubleshooting

This guide will not detail the ins and outs of Windows or IIS administration. However, a few problems may occur that can be solved easily in cases where the Fingerprint Engine service does not start or is producing an exception. This can have a number of causes, the most frequent ones are mentioned in this section.

3.7.1 Fingerprint Engine instance problems

In case the Fingerprint Engine instance does not start, the configuration of the service in IIS might be invalid. You should check the following:

- If you run into trouble running Elsevier Fingerprint Engine, check whether .Net version 4.0 has been installed.
- Check if the IIS server has been configured to run ASP/.Net applications. If ASP/.Net scripting was not installed right from the start, it can be installed by running the following from the command prompt (in administrator mode). Cf. http://msdn.microsoft.com/en-us/library/k6h9cz8h%28v=VS.100%29.aspx.

```
pushd c:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\

[Or, on 64 bit machines:]

pushd c:\WINDOWS\Microsoft.NET\Framework64\v4.0.30319\

aspnet_regiis.exe -iru
popd
```

- Make sure that the Elsevier Fingerprint Engine folder in the IIS website has been marked as an 'application' with scripting privileges. The installer performs this normally, but this might have failed. Also, from the application's ASP.Net tab (in the IIS management console for IIS6), select the ASP.Net framework 4.
- Make sure that the ServiceModel of .Net is registered (if not, see below). This registration is part of the ASP.Net system setup, and can be found
- In between steps, stop and restart the IIS application pool that contains the Elsevier Fingerprint Engine application. Once an initialisation of the Elsevier Fingerprint Engine instance fails, the error will not be recovered until re-start.
- On all operating systems, you must allow Asp.Net 4.0 as a web server extension. Some systems are not configured that way by default.

3.7.2 SQL Server connection problems

The Elsevier Fingerprint Engine IIS server process must have enough rights to be able to connect to resources it needs. Since SQL Server has a strict access policy, this is the most likely resource that cannot be connected to.

IIS runs the Fingerprint Engine service in an application pool, that has an identity. This identity is a user or a service like the local `NETWORK SERVICE` or `LOCAL SERVICE`. These should be known to SQL Server as logins and connected to a user that has access to the databases that the Fingerprint Engine instance needs as resource.

The intricacies of achieving this will not be dealt with here. See, e.g., MSDN.

3.8 Testing a Elsevier Fingerprint Engine instance

If the Elsevier Fingerprint Engine instance has successfully been installed on a host machine (say, localhost) then the IIS server should allow the Elsevier Fingerprint Engine process to run and initialize.

An initial proof of a successfull installation can be obtained by opening the following page in a web browser:

```
http://localhost/taco/TacoService.svc/          (replace 'taco' with your instance name)
```

This call should return a status report for the Elsevier Fingerprint Engine instance at hand. The resulting XML contains information on

- available workflows (relevant for the Elsevier Fingerprint Engine client - programmer)
- available annotation types (relevant for the Elsevier Fingerprint Engine users / client)
- available static object interfaces (relevant for administrator that builds a Elsevier Fingerprint Engine configuration)

See *instance configuration* for more detailed information.

3.8.1 Inspecting a Elsevier Fingerprint Engine instance

If the Elsevier Fingerprint Engine instance has successfully been installed on a host machine, an administrator or power-user can inspect the installation for what items have been installed by opening a status page with the following URL:

```
http://localhost/taco/Status.svc/          (replace 'taco' with your instance name)
```

This should return the file versions and other information on executable DLLs. Should you receive additional DLLs from Elsevier, these DLLs may required specific versions of previously installed DLLs. This status page will provide the administrator with information. Should you need to report a problem to Elsevier, also provide this status information.

CHAPTER

FOUR

CONFIGURING THE ELSEVIER FINGERPRINT ENGINE

4.1 General introduction

Configuration of the Elsevier Fingerprint Engine is done by means of a number of XML files, that will be discussed in general in this chapter. There are a number of aspects these XML files have in common. These will be discussed first.

4.1.1 Use of namespaces

All XML configuration files use XML namespaces extensively to avoid name clashes. One of those is the Extensible Application Markup Language (XAML) namespace: http://schemas.microsoft.com/winfx/2006/xaml. See MSDN for an overview of XAML.

The other ones are namespaces that refer to the various classes in the Elsevier Fingerprint Engine. They are based on the Common Language Runtime namespace mapping mechanism. These namespaces consist of two parts. One is the assembly that contains some or all of the referenced CLR namespace, the other is the class, i.e., a module providing a defined set of functionality. Namespaces that are commonly used in the configuration files are:

- clr-namespace:Collexis.Common;assembly=TACO.Core
- clr-namespace:System;assembly=mscorlib
- clr-namespace:TACO.Activities;assembly=TACO.Core
- clr-namespace:TACO.Core.Serialization;assembly=TACO.Core
- clr-namespace:TACO.Core;assembly=TACO.Core
- clr-namespace:TACO.Modules;assembly=TACO.Core
- clr-namespace:TACO.Modules;assembly=TACO.POSTagger
- clr-namespace:TACO.Normalizers;assembly=TACO.Normalizer.en
- clr-namespace:TACO.StaticObjects;assembly=TACO.Core
- clr-namespace:TACO.Tokenizers;assembly=TACO.Tokenizers

Note that in XML these namespaces are typically given a short name which is then used in the remainder of the XML file. In the example below the FilePrefixPair node is defined in the `clr-namespace:TACO.StaticObjects;assembly=TACO.Core` namespace.

```
<WorkflowRuntime Name="WorkflowRuntime"
      xmlns:xaml="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns="clr-namespace:TACO.Core;assembly=TACO.Core"
      xmlns:io="clr-namespace:Collexis.Common.IO;assembly=Collexis.Common"
      xmlns:s="clr-namespace:TACO.StaticObjects;assembly=TACO.Core">
  <WorkflowRuntime.StaticObjects>
    <NamedObjects>
      <s:PrefixPair xaml:Key="Coordinations" Ordered="false" Remainder="3" >
          <s:PrefixPair.DataSource>
```

```
              <io:CsvData Source="{io:Text {io:File Coordinations.txt}}" Headers="Column1,Column2
          </s:PrefixPair.DataSource>
      </s:PrefixPair>
      ...
    </NamedObjects>
  </WorkflowRuntime.StaticObjects>
</WorkflowRuntime>
```

4.1.2 Common attributes

The configurations involve specifying a number of different objects like activities, workflows etc. All of these objects are identified with a user-supplied identifier. For the *runtime* and *workflow list* configuration this is the `Key` attribute from the XAML namespace. For the *workflow configuration* it is the `Name` attribute from the XAML namespace.

4.2 Configuring a Fingerprint Engine instance

A Elsevier Fingerprint Engine Instance is delivered as an IIS Service. After installing the Elsevier Fingerprint Engine, there will be an installation folder, which usually is `C:\inetput\wwwroot\taco` (taco stands for the instance name). In this folder we find:

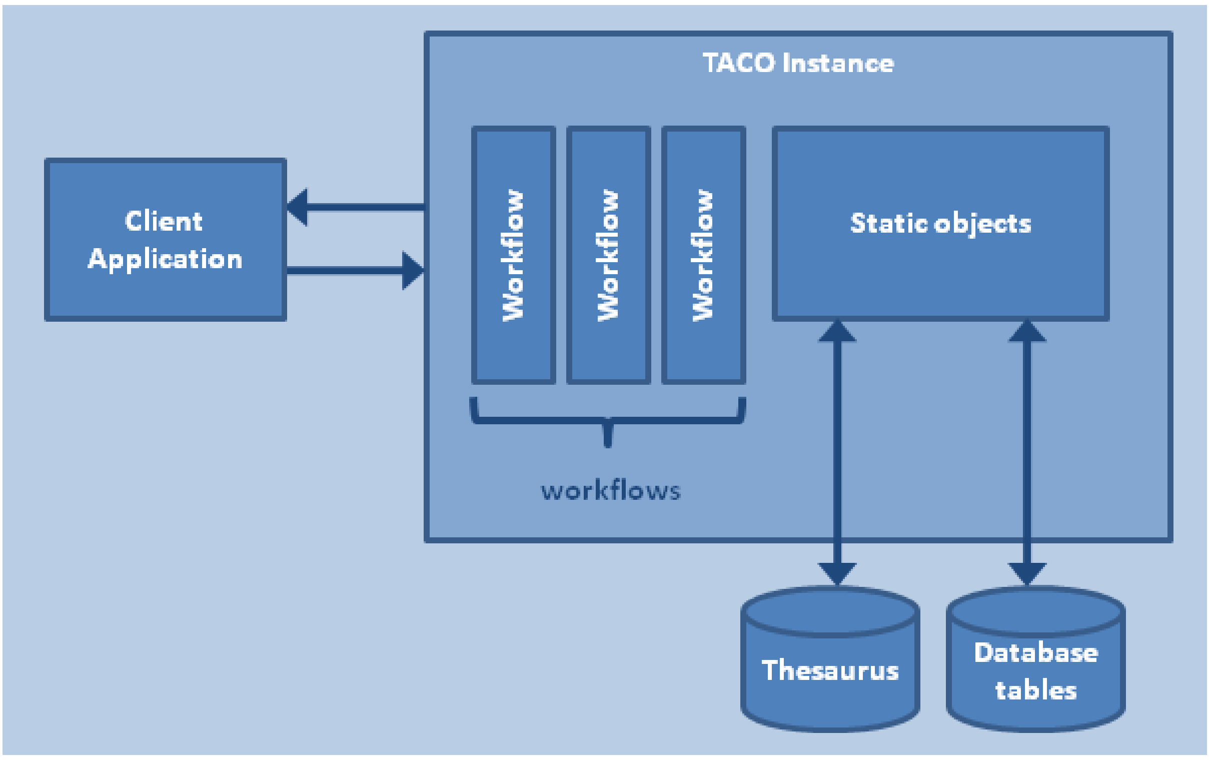
folder	contents
..\bin	containing binaries
..\config	configuration file for use with the Elsevier Fingerprint Engine program
..*	several files providing the entry to the Fingerprint Engine service

The Elsevier Fingerprint Engine instance needs several configuration parameters, in order to be able to instantiate the right functionality.

These configuration parameters are contained in a set of Xml-style files that are stored in a single configuration root folder. During installation, under the instance root folder another folder was created with the name `config`. Under this folder, some sample configurations have been provided. Should the administrator wish to define his or her own Workflow Definition, e.g. `myConfig`, then it is recommended to copy one of the folders, e.g., `config\MeSH` to `config\myConfig` and start editing the appropriate files in that new folder. Finally, the web.config file in the instance roots needs to be adapted:

```
<appSettings>
  <add key="TACOConfig" value="~/config/myConfig" />
</appSettings>
```

A Elsevier Fingerprint Engine instance consists of one or more workflows that each consist of a number of activities or processing steps. These activities need resources that may be stored in a database or files. These resources are static objects, since they do not change over the lifetime of a Fingerprint Engine instance. See the figure below.



The instance configuration as found in the `config\*` subfolders consists of the

- The *runtime configuration* found in the `RuntimeConfiguration.xaml` file;
- The *configuration of individual workflows* in one or more `*.xoml` files; and
- The *workflow list configuration* found in the `WorkflowList.xaml` file.

Several of the Elsevier Fingerprint Engine text processing steps, such as Normalization, Thesaurus-based concept finding or Part-Of-Speech tagging, may require a large amount of data in order to perform their tasks. The data for these activities is therefore stored statically, which means that the data remains loaded throughout the lifetime of the Elsevier Fingerprint Engine instance. These data are placed in *Named Objects*. Named or Static objects must be configured in the service configuration. The configuration of the Named Objects is detailed in the section on *Runtime Configuration*.

The list of workflows and the (possibly several) desired output formats for each workflow can be configured as detailed in the section on *Workflow Configuration*. Each workflow can be defined individually, see *Configuration of individual workflows*.

4.2.1 Predefining some context data before loading the Fingerprint Engine

In some instances, it may be required or necessary to define some configuration parameters before the Fingerprint Engine instance is loaded. The Fingerprint Engine provides a mechanism to define context variables - similar to environment variables - for the time of initializing the Fingerprint Engine.

For example, suppose that the location of a certain database is variable between servers - such that the connection string is different between server 1 and server 2. We may still want to use the exact same setup, but just change the connection string - which we know is referenced by the name of `ThesaurusConnectionString`.

```
Server1: ThesaurusConnectionString=Server=(local);Database=Thesaurus;Trusted_Connection=Yes;
Server2: ThesaurusConnectionString=Server=192.3.1.48;Database=Thesaurus;Trusted_Connection=Yes;
```

We may then redefine this connection string in the application configuration.

```
<appSettings>
  <add key="TACOConfig" value="~/config/myConfig" />
  <add key="ThesaurusConnectionString"
```

```
              value="Server=192.3.1.48;Database=Thesaurus;Trusted_Connection=Yes;" />
</appSettings>
```

This definition will be picked up later in the configuration.

When all configuration files are in order, the Elsevier Fingerprint Engine Instance can be started.

4.3 Configuring a workflow

Individual workflows are defined in file with the XOML Windows Workflow Markup Language format. These Workflows in the Elsevier Fingerprint Engine are *sequential* workflows. Sequential workflows are defined by a sequence of XOML-defined *activities*. Each workflow is defined in its own XOML-file, that should be given the name of the workflow at hand.

The activities are defined in a list of XOML definitions of objects. (As a technicality: these objects must all derive from `TextAnalysisWorkflow`. All available activities in a Elsevier Fingerprint Engine package are programmed that way.) At the moment most available Elsevier Fingerprint Engine activities are contained in the *TACO.Core* assembly, but activities may reside in other assemblies as well. Conveniently, all activities are located in the namespace `TACO.Activities`. In the header of the activity-XOML file we find the following header:

```
<TextAnalysisWorkflow
    xmlns="clr-namespace:TACO.Core;assembly=TACO.Core"
    xmlns:a="clr-namespace:TACO.Activities;assembly=TACO.Core"
    xmlns:p="clr-namespace:TACO.Modules;assembly=TACO.POSTagger"
    xmlns:s="clr-namespace:TACO.Core.Storage;assembly=TACO.Core"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SampleWorkflow"
    x:Name="<YOUR WORKFLOW NAME>">
```

Activities must read and write data from and to the datastructure that is passed through the workflow. Deleting data is not possible; modification of single annotations is possible.

4.4 Configuring a list of workflows

The *workflow configuration* file contains a list of defined workflows. The Elsevier Fingerprint Engine instance embedded in the IIS Server will expose handles for each workflow defined. The URLs for each workflow are based on the default host URL as defined during the Fingerprint Engine *instance installation*.

The server exposes two handles for each workflow: a SOAP interface and a REST interface. The resulting URLs for a workflow as below (SampleWorkFlow) are as follows.

```
REST: BASEURL/WorkFlowName/TacoService.svc : http://localhost/TACO/SampleWorkflow/TacoService.svc
SOAP: BASEURL/WorkFlowName/Xml.svc         : http://localhost/TACO/SampleWorkflow/Xml.svc?wsdl
```

The workflow configuration needs, for each workflow:

- a (unique) name for the workflow,
- a reference to the XOML-file that contains the *configuration of that individual workflow*,
- a reference to the static object that defines the output *annotation filter* for this workflow.
- [optional] a reference (by name) to the *preprocessor* pre-processing the input to this workflow.
- [optional] a reference (by name) to the object that defines an *annotation extender* for the outputs of this workflow.

4.5 Configuring an activity

Activities must produce output annotations based on the input text and / or annotations produced by previous activities. This implies that the ordering of activities is not trivial: it is obviously not meaningful to produce a concept fingerprint (an aggregate of found terms) if the term finding activity was not executed before; it is impossible to find terms based on normalized word forms if the 'normalize' activity has not been executed first.

All activities (such as listed below) as implemented as software classes deriving from a base class, the `TextAnalysisActivity`. Each activity must define its input and output handles. For instance, the 'normalize' activity may specify that it requires `Token` annotations and needs access to the original input text and produces normal word forms; the term finding activity may specify that it needs `Token` and `Word` annotations and produces `TermAnnotation` annotation.

The input handles for the activities are assigned to by the workflow execution engine. The designer of a workflow does not need to know specifically what these handles are if s/he follows the instructions below; for most activities it is indicated what input is required. If one puts the activities in the right order, the input handles will be assigned correctly, implicitly.

4.5.1 Input Adapters

For some activities, however, the user may need to control the input in order to achieve a specific result. For some possible use cases, the documentation of specific activities will point out how to alter or filter the input of the activity. The usual activity configuration contains the activity's name and a definition of its attributes (in Xaml syntax). The input handles for Activities are their 'hidden' attributes - that one normally does not specify. For some types of input handle, the Fingerprint Engine provides functionality to specify an adapter that controls the input to an activity.

For instance, activities requiring the input of `Sentences` implicitly take all sentences that are present in the data structure; however, using the `BlockOutTerms` adapter, one specifies that only sentences, or portions of sentences that are not known as Terms should be the input to that `Sentences` handle. For more detail, see the section on the *Block out terms adapter*.

As another example, consider the aggregation of found terms in the input text. The *Find Terms* activity annotates all terms in the text with `TermAnnotations`. These term annotations can be aggregated into a Collexis fingerprint (concept vector) of the document by the *Make Fingerprint* activity. In the most common case, terms are searched from only one thesaurus; however, it is perfectly possible to execute the *Find Terms* activity twice for different thesauri, and subsequently compile two fingerprints. To make sure that each fingerprints is based on terms from one thesauri only, we need to apply a filtering adapter (SelectTerms) in order to select terms from the right thesaurus source only.

4.6 The runtime configuration

The configuration for the Elsevier Fingerprint Engine runtime is contained in a XAML file called `RuntimeConfiguration.xaml`. The top element in this XAML file is WorkflowRuntime. The namespaces used in the runtime configuration must refer to the Elsevier Fingerprint Engine assemblies that contain the static object types used in the configuration. Most objects reside in the `TACO.Core` assembly and in similarly named `TACO.Core` namespace. The static data objects for a Elsevier Fingerprint Engine instance are placed in *Named Objects*. The following XAML code snippet provides the basic structure of the Runtime configuration. It is instructive to review the sample configuration files delivered in the Elsevier Fingerprint Engine instance installation.

```
<WorkflowRuntime Name="WorkflowRuntime"
                 xmlns:xaml="http://schemas.microsoft.com/winfx/2006/xaml"
                 xmlns="clr-namespace:TACO.Core;assembly=TACO.Core"
                 xmlns:c="clr-namespace:System;assembly=mscorlib"
                 xmlns:m="clr-namespace:TACO.Modules;assembly=TACO.Core"
                 xmlns:n="clr-namespace:TACO.Normalizers;assembly=TACO.Normalizer.en"
                 xmlns:p="clr-namespace:TACO.POSTagger;assembly=TACO.POSTagger"
                 xmlns:t="clr-namespace:TACO.Tokenizers;assembly=TACO.Tokenizers"
                 >
    <!-- static objects as a list (self-organizing by interface type) -->
    <WorkflowRuntime.StaticObjects>
        <NamedObjects>
            <OBJECTTYPE xaml:Key="NAME" OTHERATTRIBUTE1="..." OTHERATTRIBUTE2="..." />
        </NamedObjects>
    </WorkflowRuntime.StaticObjects>
</WorkflowRuntime>
```

Named objects have types (OBJECTTYPE) that reflect their implementation in the software. Multiple named objects of the same type (for instance, a *Thesaurus* or *Lexicon*) may be instantiated at the same time but then these objects must be differentiated by name, configured in the attribute `xaml:Key`.

Whatever the type of the objects, what matters to the activities calling on the objects is their interface. For instance, we may instantiate wordsets from a database table or from a file; a *database wordset* will be instantiated from a database table and a *file wordset* may be instantiated from a file, this does not matter to the activity that requires any wordset compliant with the `IWordSet` interface.

The Elsevier Fingerprint Engine runtime detects the relevant interfaces of Static Objects and will group these objects automatically. Static objects are organized *pools* containing objects with the same interface. Static objects are persistent and constant throughout the lifetime of a runtime. Since static objects are organized per interface type; each single static object must have a **Key** that is unique to the static object with that interface type, i.e. in its pool of similar static objects. Other static objects and activities can query the static objects by this **Key**.

See the *static objects section* for a more detailed description of the currently available static objects.

CHAPTER

FIVE

USING THE ELSEVIER FINGERPRINT ENGINE

5.1 The Fingerprint Engine REST service handle

If a Fingerprint Engine instance has successfully been installed on a host machine (say, localhost) then the IIS server will allow the Fingerprint Engine process to run and initialize.

There are a number of REST service access points. There are several modes of accessing indexation / text analysis services.

1. *single document [Xml or plain text] with fixed workflow*
2. *single document [Xml or plain text] with workflow defined in the parameters*
3. *multiple documents [Xml or plain text] with fixed workflow*
4. *normalizer mode*

5.2 Single document with fixed workflow

```
http://localhost/TACO/TacoService.svc/          // (replace 'TACO' with your instance name)
```

The functionality of workflows are selected by a path extension in the URL that contains the workflow name, for instance:

```
http://localhost/TACO/TacoService.svc/POSTagger                    // for POSTagger workflow

http://localhost/TACO/TacoService.svc/ThesaurusIndexing            // for the ThesaurusIndexing wor
```

If the service handle is started without path extension, it returns the *status page*. If the workflow name is specified in the extended, the client can POST plain text or xml to be processed.

If the user sends plain text, a workflow must be selected that executes without preprocessing or handling of metadata. Should a user want Xml to be processed, s/he probably wants a *preprocessor* to section the text before processing. If the user does not only want to enter a structured document but also metadata, a *preprocessor* is also required.

5.2.1 Calling the service

A Fingerprint Engine instance exposes its functionality as a REST interface or as a SOAP interface. In the Fingerprint Engine samples installation, a sample Visual Studio project is included that demonstrates how to call the Fingerprint Engine. As a service to DotNet users, a Fingerprint Client library is also provided. The essential code in that project is listed below.

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url + "/" + workflow);
request.Method = WebRequestMethods.Http.Post;
request.ContentType = "application/text";
using (StreamWriter writer = new StreamWriter(request.GetRequestStream()))
```

```
{
    writer.Write(text);
}

using (StreamReader reader = new StreamReader(request.GetResponse().GetResponseStream()))
{
    // process output
}
```

The most relevant operation mode for the Fingerprint Engine is to specify the workflow required, and send text to it. A sample `url` in the example above could be `http://localhost/TACO/TacoService.svc/MyWorkflow`. The text sent to the Fingerprint Engine is processed, and an XML stream is returned. This XML stream can be consumed by storing it in a buffer after which a DOM parser extracts the elements the client needs, or connect the stream to a SAX parser so that parsing can start before the end of the stream has been read.

For testing purposes it is also possible to use the web browser to execute a particular workflow on a (short) text. This is achieved by entering the following URL in the web browser:

```
http://localhost/taco/TacoService.svc/MyWorkflow/This is the text to process
```

So an example using one of the default workflows would be:

```
http://localhost/taco/TacoService.svc/MeSH/Distribution of malaria vaccines in Central Africa
```

The result would be a workflow response in XML format. This is described in detail in the section on the *return XML format*.

5.3 Single document with workflow defined in parameters

The Fingerprint Engine provides access to its annotation services by POST-ing Xml to a single 'Process' handle.

```
http://localhost/TACO/Process.svc/          POST xml
```

This Xml *must* follow the following structure:

```
<Xml>
    <Workflow Type="PlainText" >MeSH</Workflow>
    <Text>Text</Text>
</Xml>

<Xml>
    <Workflow>MeSHXml</Workflow>
    <Text><Title>Title</Title><Abstract>Abstract</Abstract></Text>
</Xml>
```

The Workflow node has an optional attribute `Type` that can have two values: 'Xml' or 'PlainText'. If omitted, the value defaults to *Xml*. If the value is 'PlainText', the contents of `<Text>` node will be sent to the workflow defined - if the value is 'Xml', the `<Text>` node will be sent in its entirety to the workflow defined.

5.4 Bulk Indexing

Aside from single-document indexing, a Fingerprint Engine instance also allows for indexing multiple documents in a single call. Due to restrictions on the size of the data that the host process (the IIS service) can take, more than 100 documents (depending on size) would hardly be feasible, but using bulk indexing may ease the process on the client side. The bulk indexing is accessible only via a REST interface.

The ReST interface that accepts bulk indexing calls is present at the base address of the Fingerprint Engine, extended by `Bulk.svc`. For accessing the bulk indexing service, three arguments need to be appended to the ReST interface path, which are

- workflow
- record marker
- ID marker

A sample URL could be

```
http://my.ip.address//EFE7400/Bulk.svc/MeSH/Document/ID/
```

The records to be indexed must be POSTed to the URL. The stream of records must be valid Xml with at least two levels:

- A root node [any]
- A record wrapper that signals a new record. The name of the record wrapper (XmlNode) is the second element in the path above ['Document'].

Each record in the stream must at least have an ID marker that identifies the document in the context of the stream. If this ID is not present, an exception will be thrown. The ID must be unique in the stream. The ID may either be encoded in an embedded node as text value, or passed as an attribute to the record marker node.

The Xml embedded in each document is subject to identical requirements as the xml that is sent to the Fingerprint Engine in individual calls. Xml may be embedded [if an Xml-capable workflow is selected], as well as metadata.

After all documents have been processed, a single Xml stream is returned. The Xml root node and document wrapper node are the same as those from the input.

As with the normal single-document text processing, there are essentially two modes of processing, depending on the *preprocessor*.

- Plain text
- Xml sectioned documents

When sending wrapped documents to an Xml-enabled sectioned document processor, the document wrapper node is *included* in the data sent to the Fingerprint Engine. When sending wrapped documents to a plain text processor, the document wrapper is not included. The bulk indexing handle 'knows' what to do with the document wrapper by inspecting the incoming Xml. If the ID is in an attribute to the document wrapper node, the bulk indexer sends the contained text (inferring that the targeted workflow is a plain text workflow); if the ID is wrapped in a separate node, the indexer sends the document wrapper nodes and everything in it.

```
<Document ID="1234">this is sent to the plain text processor</Document>

<Document>
    <ID>1234</ID>
    <Title>Title text</Title>
    <Text>All of this Xml is sent to the processor</Text>
</Document>
```

So the bulk indexer always does The Right Thing. The result is a set of documents following the rules of the *return XML format*.

5.5 Return XML format

The format of the XML stream returned by the Fingerprint Engine is quite general. A Fingerprint Engine workflow adds annotations to the text, as defined by the *OutputFilter*. The exact list of output annotations is also listed in the *status xml information of the workflow*.

The following XML snippet shows the structure of the XML file. It is a portion of the result XML from the Fingerprint Engine after sending the input text "this is a sample text".

```
<TextAnalysis xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
              xmlns="http://www.collexis.com/annotations/" >
 <Annotations xmlns:d2p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
```

```
  <Annotation i:type="Token">
   <End>4</End>
   <Offset>0</Offset>
   <Capitalisation>Lowers</Capitalisation>
   <Type>Alpha HasWhitespaceLeft HasWhitespaceRight</Type>
  </Annotation>
  ...
 </Annotations>
 <Text>this is a sample text</Text>
</TextAnalysis>
```

The top element in the XML is `TextAnalysis`. On the second level there are two nodes: `Annotations` and `Text`. The `Text` contains the text sent. The `Annotations` node contains a list of `Annotation` nodes.

Each `Annotation` node has a `type` attribute that specifies the annotation type. The nodes enclosed in the `Annotation` node depend on the type of annotation represented. The types of annotation that can be returned can be queried by opening the *status page*. For each workflow, the annotations returned are listed in the `OutputAnnotations`. The annotations returned from each workflow can be controlled by an *output filter* (see also below: *What annotations to expect*).

Each annotation node has the following structure:

```
<Annotation i:type="TYPE">
  <PropertyA>value</PropertyA>
  <PropertyB>value</PropertyB>
  ...
</Annotation>
```

What properties are returned is specific for the *annotation type*. Annotation fields can be changed, deleted or added by *annotation extenders*.

5.6 Normalization Handle

Should you just be interested in the normalization capabilities of the Fingerprint Engine, then use the normalize handle.

```
http://localhost/taco/Normalize.svc/{Normalizer}        POST
```

Note! that a static object that embodies a normalizer of the name {Normalizer} must be defined in the runtime configuration of the Fingerprint Engine instance at hand. The Xml to be posted should look like this:

```
<Xml><Text id="1">text to normalise</Text><Text id="2">behavioural changes</Text></Xml>
```

The Xml returned will look like this:

```
<Xml><Text id="1">text to normalize</Text><Text id="2">behavioral change</Text></Xml>
```

Basically, the input xml is sent to the output, except that all text has been normalized.

5.7 The Fingerprint Engine status information

Requesting the base URL for REST services reports the status of the Fingerprint Engine in XML format. The status report can be easily viewed in a web browser. The status report shows the configurations, types and workflows defined for the Fingerprint Engine instance at hand. The report is in XML format and describes the functionality that the Fingerprint Engine instance is providing. The example below shows its general structure:

```
<Status>
  <Workflows>
    <Workflow name="MyWorkflow">
      ...
    </Workflow>
    <Workflow name="MySecondWorkflow">
      ...
    </Workflow>
    ...
  </Workflows>
  <AnnotationInterfaces>
    <Interface name="TACO.Data.IConceptAnnotation"/>
    ...
  </AnnotationInterfaces>
  <AnnotationTypes>
    <Type name="TACO.Data.Token" >
    ...
    </Type>
    ...
  </AnnotationTypes>
  <StaticObjectInterfaces>
    <Interface name="TACO.Core.IWordSet"/>
    ...
  </StaticObjectInterfaces>
  <Activities>
    <Activity name="TACO.Activities.Tokenize">
    ...
    </Activity>
    ...
  </Activities>
</Status>
```

The status report contains a list of the following elements:

- *available workflows*. These are relevant for the Fingerprint Engine client-programmer.
- *annotation interfaces*. These are relevant for the Fingerprint Engine users / client.
- *annotation types*. These are relevant for the Fingerprint Engine client-programmer.
- *static object interfaces*. These are relevant for administrators that build a Fingerprint Engine configuration.
- *activities* These are relevant for administrators that build a Fingerprint Engine configuration.

The annotation types that are returned can be filtered by applying an *annotation filter*; annotation filters are defined in the runtime configuration, and referenced in the *work flow list*. It is a usual scenario to have a sequence of activities defined once in a *work flow configuration*, while several workflows based on that sequence are exposed, different only by the annotation filter.

5.7.1 Workflow status information

An example status report of a particular workflow is shown below:

```
<Workflow name="MeSH2011Concepts">
  <Annotations>
    <Annotation name="TACO.Data.ConceptAnnotation"/>
    <Annotation name="TACO.Data.TermAnnotation"/>
    <Annotation name="TACO.Data.EmptyExpansion"/>
    <Annotation name="TACO.Data.TokenPattern"/>
    <Annotation name="TACO.Data.Expansion"/>
    <Annotation name="TACO.Data.Abbreviation"/>
    <Annotation name="TACO.Data.ArrayExpansion"/>
    <Annotation name="TACO.Data.NormedWord"/>
    <Annotation name="TACO.Data.MultiTokenWord"/>
    <Annotation name="TACO.Data.Word"/>
    <Annotation name="TACO.Data.AllCapsRegion"/>
    <Annotation name="TACO.Data.Token"/>
    <Annotation name="TACO.Data.Sentence"/>
  </Annotations>
  <OutputAnnotations>
    <Annotation name="TACO.Data.ConceptAnnotation"/>
  </OutputAnnotations>
</Workflow>
```

The workflow name is denoted in the `name` attribute of the workflow element. The `Annotation` elements contain all annotations that are created at some point in the workflow. The `OutputAnnotation` elements contain all annotations that are returned by the workflow. This is always a subset of the annotations listed in the `Annotation` elements, with the exception of section annotations that are produced by an optional *preprocessor*. The *OutputFilter* configuration of the workflow defines which annotations are returned by the workflow.

5.7.2 Annotation interfaces status information

Annotation interfaces list the interfaces that are used in the instance. Note that this is a different list than the one shown in the `StaticObjectInterfaces` node. The `StaticObjectInterfaces` can be regarded as interfaces that provide functionality to perform a specific function or provide a basic datastructure, while the `AnnotationInterfaces` provide data interfaces for the annotations. The `Output` nodes describing the activities refer to annotation interfaces from this list.

```
<AnnotationInterfaces>
  <Interface name="TACO.Data.IAbbreviation"/>
  <Interface name="TACO.Data.IConceptAnnotation"/>
  <Interface name="TACO.Data.IExpansion"/>
  <Interface name="TACO.Data.ILanguageAnnotation"/>
  <Interface name="TACO.Data.IPOSTag"/>
  <Interface name="TACO.Data.IRegion"/>
  <Interface name="TACO.Data.IScoredAnnotation"/>
  <Interface name="TACO.Data.ISentence"/>
  <Interface name="TACO.Data.ITermAnnotation"/>
  <Interface name="TACO.Data.ITextRange"/>
  <Interface name="TACO.Data.IToken"/>
  <Interface name="TACO.Data.ITokenPattern"/>
  <Interface name="TACO.Data.ITokenRange"/>
  <Interface name="TACO.Data.IWord"/>
  <Interface name="TACO.POSTagger.ITagProbabilityDistribution"/>
</AnnotationInterfaces>
```

5.7.3 Annotation types status information

The `AnnotationTypes` node describes the annotation types used in the Fingerprint Engine instance. Each type is described with its name, and a list of the interfaces it implements. In the example below the `TACO.Data.Token` type implements `TACO.Data.ITextRange` and `TACO.Data.IToken`. Note that each of the interfaces mentioned is present in the list of annotations in the `AnnotationInterfaces` node. The `indexType` attribute is for internal purposes only. The `Input` nodes describing the activities refer to annotation types from this list.

```
<AnnotationTypes>
  <Type name="TACO.Data.Token" indexType="TACO.Data.RangeAnnotationIndex`1">
    <Implements>
      <Interface name="TACO.Data.ITextRange"/>
      <Interface name="TACO.Data.IToken"/>
    </Implements>
  </Type>
  ...
</AnnotationTypes>
```

5.7.4 Static objects status information

The `StaticObjectInterfaces` node lists the static object interfaces that are used in the instance. Note that this is a different list than the one shown in the `AnnotationInterfaces` node. The `StaticObjectInterfaces` can be regarded as interfaces that provide functionality to perform a specific function or provide a basic datastructure, while the `AnnotationInterfaces` provide data interfaces for the annotations.

```
<StaticObjectInterfaces>
    <Interface name="TACO.Core.ITokenizer"/>
    <Interface name="TACO.Core.INormalizer"/>
    <Interface name="TACO.Core.IWordSet"/>
    <Interface name="TACO.Modules.ITokenPatternMatcher`1"/>
    <Interface name="TACO.Modules.ISimpleTokenPatternMatcher"/>
    <Interface name="TACO.Modules.ITokenPatternMatcher"/>
    <Interface name="TACO.Modules.ITokenPatternAnnotator"/>
    <Interface name="TACO.Core.IStringMap"/>
    <Interface name="TACO.Core.IMultiNormalizer"/>
    <Interface name="TACO.Modules.IThesaurus"/>
    <Interface name="TACO.Modules.IExtendedWordSet"/>
    <Interface name="TACO.Core.Serialization.IAnnotationExtender"/>
    <Interface name="TACO.Core.Serialization.IOutputFilter"/>
    <Interface name="TACO.Core.ILanguageDetector"/>
    <Interface name="TACO.Core.IPatternAnnotator"/>
    <Interface name="TACO.Core.IPOSTagger"/>
    <Interface name="TACO.Data.IAnnotationIndexesToListConverter"/>
    <Interface name="TACO.Modules.IPOSPatternMatcher"/>
    <Interface name="TACO.StaticObjects.IPrefixPair"/>
    <Interface name="TACO.Modules.ILanguageModel"/>
</StaticObjectInterfaces>
```

5.7.5 Activities status information

The `Activity` elements describe the activities that are active in the instance in one or more workflows. Below is an example of a few activities.

```
<Activities>
  <Activity name="TACO.Activities.Tokenize">
    <Input>
      <Property name="Text" type="System.String" />
      <Property name="Sections" annotationType="TACO.Data.ISection" indexType="TACO.Data.INamedAnn
    </Input>
    <Output>
      <Property name="Tokens" annotationType="TACO.Data.Token" indexType="TACO.Core.Storage.IAnnot
      <Property name="Words" annotationType="TACO.Data.Word" indexType="TACO.Core.Storage.IAnnotat
      <Property name="Sentences" annotationType="TACO.Data.Sentence" indexType="TACO.Core.Storage
    </Output>
  </Activity>
  <Activity name="TACO.Activities.Normalize">
    <Input>
      <Property name="Tokens" annotationType="TACO.Data.IToken"
                indexType="TACO.Data.IIndexBasedAnnotationIndex`1" />
      <Property name="Words" annotationType="TACO.Data.IWord"
                indexType="TACO.Data.IAnnotationXYIndex`1" />
    </Input>
    <Output>
      <Property name="WordStorage" annotationType="TACO.Data.NormedWord" indexType="TACO.Core.Sto
      <Property name="MultiTokenWordStorage" annotationType="TACO.Data.MultiTokenWord" indexType=
    </Output>
  </Activity>
  <Activity name="TACO.Activities.MarkupWords">
```

```
    <Input>
      <Property name="WordIndex" annotationType="TACO.Data.IWord"
                indexType="TACO.Data.IAnnotationIndex`1" modified="true" />
    </Input>
    <Output/>
  </Activity>
</Activities>
```

Each `Activity` element has an `Input` node and an `Output` node. The `Input` node lists the requirements of the input text. For activities that do not require annotations and can process unannotated text the input is described as

```
<Property name="Text" type="System.String"/>
```

For activities that do require annotations, these annotations are listed in the `Input` node. For instance, the `Normalize` activity needs annotations that implement annotation interfaces *TACO.Data.IToken* and *TACO.Data.IWord*.

Input annotations that are optional, have an additional attribute `optional="true"`. See the example for `Tokenize` above. For activities that modify an input annotation, an additional attribute `modified="true"` is added. In these cases the `Output` node may be empty.

Activities that do not require input have an empty `Input` node. Such activities can only set global annotations. These could be used, for instance, as triggers for other activities later in the workflow.

The annotations the activity produces are listed in the `Output` node. The `Tokenize` activity in the example above outputs three annotations of the types `TACO.Data.Token`, `TACO.Data.Word`, `TACO.Data.Sentence`. Note that the `Input` nodes list annotation types, whereas the `Output` nodes list annotation interfaces. The *annotation type* status information describes which annotation types implement which annotation interfaces.

Activities that contain empty `Output` nodes not necessarily generate no output. It is likely that such activities modify one of the annotations listed in the `Input` node, e.g., adding or modifying a property. An example of such an activity is the `MarkupWords` activity. See also the `modified` attribute.

5.8 Error reporting

The Fingerprint Engine was designed in a way such that the likelihood of errors is minimal; in other words, errors are highly improbable. Of course there may be user settings or configurations that are wrong. The Fingerprint Engine will detect and report those issues during loading of the *status page*. Once the status page has loaded, there are no more errors to be expected. Users may experience problems such as other output than expected but in all cases, output complies with the *return Xml format* as defined above.

On unexpected and really bad errors, however, the Fingerprint Engine may internally throw an exception. This exception is caught and reported to the end user *instead of the usual data* in xml following the following syntax:

```
<error>
  <message>what went wrong</message>
</error>
```

Should this contingency occur, contact Elsevier, reporting the workflow and data that caused the error.

5.9 Testing the Fingerprint Engine

CHAPTER

SIX

ACTIVITIES

Activities are text processing steps performing one particular task. Workflows are constructed by defining a series of activities to be performed sequentially.

In this chapter all available activities are described in more detail. The activities are listed in random order, and the composer of a workflow must take care to order them correctly. If the dependencies are not correctly set, the Fingerprint Engine will throw an exception noting the deficiencies during initialisation. This exception highlights the problem.

Add non capital alternatives For each word found in an all-caps region, this function inserts a lower case alternative word form.

Annotate Patterns The `AnnotatePatterns` activity annotates patterns. These patterns may be defined in terms of regular expressions on the character level or regular expressions in terms of TokenTypes. For patterns defined in terms of PoS-tags or terms, see other pattern matchers.

Annotate POS Patterns The POS pattern annotation activity was designed for annotating patterns such as noun phrases, specific verb patterns or other patterns of elements of certain parts-of-speech.

Annotate Terms See Annotate Terms, below.

Annotate TokenRanges See Annotate TokenRange Patterns, below.

Annotate TokenRange Patterns These Annotate TokenRange... activities annotate sequences of mixed types of annotations, for instance POSTags, Words and TermAnnotations. These activities only differ in the type of output they produce. A word of warning: The definition of these patterns is an expert job.

Dehyphenate Dehyphenate: provide alternative word forms for hyphenated input text.

Detect Abbreviations Detect abbreviations activity: detects abbreviations.

Detect Capitalisation Regions Detect regions of text that are in all caps. Such regions should be dealt with separately since capitalization has a different meaning.

Detect Language Detect Language activity detects the language of the input text.

Detect Sentences Detect Sentences splits the input text and tokens into sentences.

Distribute Term Flags Distribute Term Flags spreads disambiguation information or other flags set on terms to other terms with the same identity in the same document or in a well-defined area of the document.

Expand Abbreviations Expands abbreviations previously detected.

Expand Concept Coordinations Expand coordinations of concepts that are found in the text.

Expand Coordinations Expand simple coordinations that are found in the text.

Flag Terms Flag terms that have been found by the *term finding* activity.

Find Terms The term finding activity annotates terms, based on a thesaurus.

Hide Patterns The activity 'hide patterns' uses previously detected patterns (token ranges) and inserts zero expansions corresponding to the patterns selected. Annotators that are sensitive to expansions (such as concept annotation) will then ignore the text annotated by the selected patterns.

Make Fingerprint The fingerprint making activity aggregates found concepts into concept annotations, that reflect, per concept, the relevance of the concept to the document. Put in other words, the output of the Find Terms activity is aggregated to a *document vector*.

Mark up Words Mark up activity marks words with a certain feature.

MultiNormalize The MultiNormalize activity can add alternative word forms for ranges of tokens, e.g. normalize *city(ies)* to *city*.

Normalize The Normalize activity computes alternative, normal word forms for input words. Typically, normalization of this type involves removal of inflectional affixes, e.g. *children* -> *child*.

Tag Parts of Speech The part-of-speech tagging activity adds part-of-speech tags to stretches of tokens that these parts-of-speech apply to. These tags do not necessarily match single tokens. but must respect token boundaries (i.e. cannot apply to a stretch of text ending or starting halfway in a token).

Tokenize The Tokenize activity splits the text in Tokens, and identifies the Words; that is, for non-whitespace tokens, base Word forms are also inserted.

6.1 Activity input and output

Activities need as input a text. Most activities need in addition one or more annotations, and most produce additional annotations. The activity descriptions below list both the input annotations and output annotations. The *input* annotations are defined using *annotation interfaces*, to provide more flexibility in creating workflows. The *output* annotations are defined using *annotation types*. All annotation interfaces and types originate from the same assembly TACO.Data.

6.2 Add non capital alternatives

The `AddNonCapitalAlternatives` activity iterates over all words in text regions that are apparently capitalized regions (for instance, titles). The activity adds lowercase alternatives to words in the output. These alternatives are considered during normalization and term matching.

It is required to *detect all-capital regions* before executing this activity.

6.2.1 Instantiation

```
<a:AddNonCapitalAlternatives       x:Name="ANCA" />
```

The activity needs no further configuration.

6.2.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Regions" annotationType="TACO.Data.IRegion" />
  <Property name="Words" annotationType="TACO.Data.IWord" />
</Input>
<Output>
  <Property name="WordStorage" annotationType="TACO.Data.NormedWord" />
  <Property name="MultiTokenWordStorage" annotationType="TACO.Data.MultiTokenWord" />
</Output>
```

6.2.3 Example

When processing the text `THIS IS AN ALL CAPS SENTENCE THAT IS LONG ENOUGH TO MEET THE MINIMUMSPAN OF THE DETECTCAPITALISATIONREGIONS ACTIVITY` this activity will produce for all words found a lowercase alternative.

6.3 Annotate Patterns

The AnnotatePatterns activity produces `TokenPattern` annotations based on the text input only. Token Patterns are purely text-based patterns, with the restriction that the boundaries of each pattern always coincide with Token boundaries, and possibly [if so required by the annotator] with sentence boundaries.

The AnnotatePatterns activity must reference a static object that complies with the `IPatternAnnotator` interface. Static objects that expose the `IPatternAnnotator` interface need the input string and token structure as input, and can output `TokenPattern` annotations of various sorts. For an overview of the functionality of the AnnotatePatterns activity, one should refer to the documentation of the different *PatternAnnotator static objects*, such as the *regular expression* token pattern matcher or the named *regular expression* pattern matcher.

The output annotation type of the Annotate Patterns activity, `TokenPattern`, have a `Name` property that is set by the Annotator object. Further activities can reference that name.

There are a number of examples of this activity in the default MeSH workflow that for instance use regular expressions to find references to Figures or other publications. Another example is a regular expression based chemical detector.

6.3.1 Instantiation

```
<a:AnnotatePatterns     x:Name="Chemicals"     Annotator="chemicals" />
<a:AnnotatePatterns     x:Name="IgnoreEmail"   Annotator="Email" />
```

6.3.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="SentenceIndex" annotationType="TACO.Data.ISentence" />
  <Property name="Text" type="System.String"/>
</Input>
<Output>
  <Property name="TokenPatterns" annotationType="TACO.Data.TokenPattern" />
</Output>
```

6.3.3 Example

6.4 Annotate POS Patterns

The `AnnotatePOSPatterns` activity annotates patterns of parts-of-speech. Of course, parts-of-speech must be tagged beforehand by the *part-of-speech tagging* activity. The annotator of POS Patterns needs a reference to a *part-of-speech pattern definition* in the static objects. The general syntax of POS Patterns can be found in a separate section on *Pattern Matching Expressions*.

6.4.1 Instantiation

```
<a:AnnotatePOSPatterns      x:Name="NPs"        Annotator="NPs" />
```

6.4.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="SentenceIndex" annotationType="TACO.Data.ISentence" />
  <Property name="POSTags" annotationType="TACO.Data.IPOSTag" />
  <Property name="Text" type="System.String"/>
</Input>
<Output>
  <Property name="TokenPatterns" annotationType="TACO.Data.TokenPattern" />
</Output>
```

6.4.3 Example

6.5 Annotate TokenRange Patterns

The Annotate TokenRange Patterns activity can annotate sequences of mixed types, for instance POSTags, Words and TermAnnotations. A word of warning: The definition of these patterns is an expert job.

Patterns are defined in a *TokenRange Regular expression*. At that location, more information on defining patterns can be found.

The token range pattern annotation activity yields annotations of the type `RangeAnnotation`. This annotation type may contain a wealth of data, e.g. internal structure of the pattern found (`Members`) etc. This is a typical expert output type that may be used if the output should be available for further expert processing.

6.6 Annotate TokenRanges

However, for use in further processing we may resort to a simpler output type, viz. the *TokenPattern type*. Using a complex token range pattern we may single out particular pattern subsections and annotate these with `TokenPatterns`s of a particular name. For instance, this type can serve as input for the HidePatterns activity that can then rule out specific portions of text from inspection by the FindTerms activity.

6.6.1 Instantiation

```
<!-- relation extraction -->
<a:AnnotateTokenRangePatterns   x:Name="TokenRanges"
                                Annotator="Relations" />

<!-- blocking specific pieces of text -->
<a:AnnotateTokenRanges          x:Name="AdverbialConstituents"
                                Annotator="AdverbialConstituents" NameProperty="name" />
```

6.6.2 Input Annotation Interfaces and Output Annotation Types

Input Annotation Interfaces are the same for both annotation activities.

```
<Input>
  <Property name="Text"           type="System.String"/>
  <Property name="TokenIndex"     annotationType="TACO.Data.IToken"       />
  <Property name="SentenceIndex"  annotationType="TACO.Data.ISentence"    />
  <Property name="TokenRanges"    annotationType="TACO.Data.ITokenRange"  />
  <Property name="Expansions"     annotationType="TACO.Data.IExpansion"   />
</Input>
```

However, the output of `AnnotateTokenRanges` is `TokenPattern` whereas `AnnotateTokenRangePatterns` yields `RangeAnnotations`.

```
<Output>
  <Property name="TokenPatterns"  annotationType="TACO.Data.TokenPattern" />
</Output>

<Output>
  <Property name="TokenPatterns"  annotationType="TACO.Data.RangeAnnotation" />
</Output>
```

6.6.3 Example

6.7 Dehyphenate

The dehyphenate activity creates alternative word forms for hyphenated input. This activity is useful for handling text that has been scanned from typeset text, or where hyphens have been inserted from other sources. The dehyphenation algorithm works in three very simple steps:

- scan for the token pattern ‘word hyphen whitespace word’ : (e.g. “dehyph- enation”, “automat- ically”)
- search for the first and last word halves of the (potentially) hyphenated form (e.g. “dehyph” and “enation”); if either of these is not found, then the full form is looked up. If the full form (“dehyphenation”) is found, the last step is conducted:
- insert an alternative `MultiTokenWord` annotation (which is comparable to ‘normalized forms’ as inserted by the *Normalize* activity.

The look-up step of the Dehyphenate activity must refer to a word table. Useful word tables are found in either a thesaurus or normalizer. The look-up tables must be named by the attribute “Words”. Executing the dehyphenation activity will result in better results during *term finding*, because after dehyphenation the original reading of the text may be available.

It is important to notice that the tables that Dehyphenation refers to are crucial to the performance of the activity. Normalizer tables are quite extensive word lists, but may miss domain specific words. Thesaurus tables will be domain specific but smaller. It is also possible to define your own *word sets* to control dehyphenation, using the usual wordset definitions in the runtime configuration.

The Dehyphenate activity must be executed *before* the Normalize activity.

By default, the dehyphenation will check every word-hyphen-whitespace-word pattern. It is possible to restrict the dehyphenator to word-hyphen-newline-word patterns by setting the attribute `NewlineRequired` to true. (If this attribute is missing it defaults to false).

6.7.1 Instantiation

```
<a:Dehyphenate    x:Name="Dehyph" Words="en" NewlineRequired="false" />
```

6.7.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="WordIndex" annotationType="TACO.Data.IWord" />
  <Property name="SentenceIndex" annotationType="TACO.Data.ISentence" />
  <Property name="Text" type="System.String"/>
</Input>
<Output>
  <Property name="DehyphenatedWords" annotationType="TACO.Data.MultiTokenWord" />
</Output>
```

6.7.3 Example

6.8 Detect Abbreviations

The detect abbreviations activity detects abbreviations of the type “Blood Group (BG)”. As a result, abbreviation annotations are added to the datastructure.

- `Abbreviation` an annotation containing the short- and long-form representation of an abbreviation; includes all this abbreviation’s attestations.

The abbreviation detection (AD) process is aware of plural forms, which enables it to recognize “POWs” as an abbreviation for “Prisoners of War”. For this, the AD needs to look up word forms that are marked as ‘Plural’. This marking is done during the normalization process; the normalizer knows of ‘plural’ and ‘singular’ forms. To select the right word forms, the AD needs to refer to the normalizer that has assigned the flags to a word annotation. The normalizer used must be named in the ‘Normalizer’ parameter in order for the abbreviation detection to use the relevant normalized word forms. As the abbreviation detector needs normalizations as input, it must be executed after the appropriate `Normalize` activity.

The abbreviation detector detects the following patterns of abbreviation definitions:

- Long Form (LF)
- SF (Short Form)
- (AD, Abbreviation Definition)

The abbreviation detector contains many detailed procedures to identify the right abbreviation and to delimit the abbreviation’s scope. These procedures include a scoring mechanism to optimally match letters in the short-form with words in the long form, and allowing for intervening function words [language sensitive; for English, for instance, ‘of’, ‘the’, etc] in the long form. These procedures are controlled by a reference to language-specific *abbreviation settings*.

The DetectAbbreviations activity has an extra [boolean] parameter called `HandleDiacritics`. If set to ‘True’, DetectAbbreviations will find IMC as an abbreviation in “índice de masa corporal (IMC)”. This flag is recommended to set to ‘true’ for languages featuring a lot of diacritics [Spanish].

6.8.1 Instantiation

```
<a:DetectAbbreviations    x:Name="Abbr" Normalizer="en"
                                        AbbreviationSettings="AbbrevEn" />
```

6.8.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="WordIndex" annotationType="TACO.Data.IWord" />
  <Property name="PatternIndex" annotationType="TACO.Data.ITokenPattern" />
  <Property name="AbbreviationIndex" annotationType="TACO.Data.IAbbreviation" />
  <Property name="SentenceIndex" annotationType="TACO.Data.ISentence" />
  <Property name="Text" type="System.String"/>
</Input>
<Output>
  <Property name="AbbreviationStorage" annotationType="TACO.Data.Abbreviation" />
  <Property name="PatternStorage" annotationType="TACO.Data.TokenPattern" />
</Output>
```

6.8.3 Example

6.9 Detect Capitalisation Regions

The `DetectCapitalisationRegions` activity scans for stretches of text that contain no lower-case characters. If the ratio of capital characters in such a region surpasses a certain threshold, these regions are designated as an `AllCapsRegion` and special rules may apply - most significantly, casing can be considered irrelevant for this particular region.

6.9.1 Instantiation

```
<a:DetectCapitalisationRegions  x:Name="ANCA"
                                    MinimumCapsRatio="50"
                                    MinimumSpan="30" />
```

6.9.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Text" type="System.String"/>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
</Input>
<Output>
  <Property name="AllCapsRegions" annotationType="TACO.Data.AllCapsRegion" />
</Output>
```

6.9.3 Example

6.10 Detect Language

The Detect language activity tries to determine the language of an input text. It needs a reference to a *language detection static object* in order to be able to do its task. The DetectLanguage activity adds one single annotation of type `LanguageAnnotation` to the text analysis datastructure.

6.10.1 Instantiation

```
<a:DetectLanguage         x:Name="LR"         Key="LR" />
```

6.10.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Text" type="System.String"/>
</Input>
<Output>
  <Property name="Language" annotationType="TACO.Data.LanguageAnnotation" />
</Output>
```

6.10.3 Example

6.11 Expand Abbreviations

The expand abbreviations activity searches for abbreviations in the text. This activity needs abbreviation annotations as input, in order to know what text portions to replace. For all abbreviations found, a replacement expansion annotation is inserted, such that “BG” can be read as “Blood Group”. Locations that define a short form, on the other hand, are replaced by an empty expansion, such that these text portion are effectively removed. This brings the benefit that an abbreviation that is defined in a text is not mistaken for a competing thesaurus term.

The following annotations are inserted:

- `Expansion` (interface IExpansion) *expansions* of short forms by their expanded long form equivalents, i.e. “BG” is replaced by “Blood Group”
- `EmptyExpansion` (interface IExpansion) deletion of short forms that are defining long forms, i.e. in the input string “Blood Group (BG)”, “(BG)” is removed.

6.11.1 Instantiation

```
<a:ExpandAbbreviations    x:Name="Abbr"  />
```

6.11.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Text" type="System.String"/>
  <Property name="Abbreviations" annotationType="TACO.Data.IAbbreviation" />
  <Property name="Tokens" annotationType="TACO.Data.IToken" />
  <Property name="Words" annotationType="TACO.Data.IWord" />
  <Property name="Expansions" annotationType="TACO.Data.IExpansion" />
</Input>
<Output>
  <Property name="EmptyExpansions" annotationType="TACO.Data.EmptyExpansion" />
  <Property name="AbbrevExpansions" annotationType="TACO.Data.Expansion" />
</Output>
```

6.11.3 Example

6.12 Expand Coordinations

Expanding coordinations is a very complicated operation. More details on the functionality of this operation can be found in the specific documentation on the subject. In any case, the `ExpandCoordinations` activity needs a reference to the following static objects:

- `WordPairs` a list of *Word pairs* that combine into coordinations (‘hetero- and homo-‘, ‘allo- and xeno-‘, etc.)

- `Coordinators` a *Word set* containing all coordinators that function as such in the language, (e.g. 'and', 'or', '&' etc.)

Combining these sources, the ExpandCoordinations activity outputs word forms (the `Word` annotation type) and expansions (the `Expansion` annotation type) representing alternative readings for the input text. The alternative readings are ranked with a score, computed relative to the score that the unexpanded forms have (1 by default). If the score factor is larger than 1, expanded forms take precedence over unexpanded forms during term finding; if the score factor is smaller than 1, unexpanded forms take precedence. (A score bigger than 1 is recommended).

Coordination expansion of this type can be executed before normalization; the forms to expand can be identified on the basis of the textual appearance. The benefit of execution expansion at an early stage is that expanded word forms will also be included in the normalization: If the pattern "allo- and xenopathies" is expanded to "allopathies and xenopathies", then the normalizer will add "allopathy and xenopathy".

6.12.1 Instantiation

```
<a:ExpandCoordinations    x:Name="EC"
                          WordPairs="Coordinations" Coordinators="Coordinators"
                          ScoreFactor="1.1" />
```

6.12.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="WordIndex" annotationType="TACO.Data.IWord" />
  <Property name="SentenceIndex" annotationType="TACO.Data.ISentence" />
  <Property name="Text" type="System.String"/>
</Input>
<Output>
  <Property name="Expansions" annotationType="TACO.Data.ArrayExpansion" />
  <Property name="ExpandedWords" annotationType="TACO.Data.Word" />
</Output>
```

6.12.3 Example

6.13 Expand Concept Coordinations

The Elsevier Fingerprint Engine Engine offers an analytical activity that can expand concept coordinations of the following pattern.

- "Lung and Breast Cancer" will be expanded to "Lung cancer and Breast Cancer".

This expansion of concept coordinations is based on a thesaurus: expansions are only validated if both parts occur in the thesaurus specified. This limitation is necessary to avoid overgeneration of expansions. The thesaurus referenced in this expansion activity does not need to be the same as in term finding later on.

The concept coordination detection searches for patterns in the input text that indicate that a certain range of tokens might be a coordination of an expandable type.

By default, the concept coordination detection only matches patterns of the type "endothelial and epithelial structures". A structure of that type is a so-called 'modifier coordination', in which two noun phrases are written by short-hand, leaving out the so-called 'head' of the noun phrase out in the first noun phrase. The adjectives 'endothelial' and 'epithelial' are the modifiers, while the word 'structures' (in the example above) is the head. An expansion of this structure then reads "endothelial structures and epithelial structures". In the original, only the modifiers were coordinated (hence the name 'modifier coordination'. In resolving the full form, the 'head' is copied to the left hand member.

Additionally, two more token patterns can be matched. The second pattern that can be expanded is the pattern of 'head-coordination'. Examples of this pattern are 'cell division and survival' or 'child health and development'. In these patterns, there is only one modifier ('cell' and 'child') which can be distributed over two noun phrases. The proper reconstructions of these noun phrases read 'cell division and cell survival' or 'child health and child development'.

The third pattern that can be expanded is the comma-separated modifier coordination, i.e. a coordination involving more than two coordinates. An example of this is 'health, life and disability insurance', which can be expanded to 'health insurance, life insurance and disability insurance'.

Each of these patterns are evaluated by the ExpandConceptCoordinations activity. This activity performs expansion of the proposed structures only if each expanded noun phrase is present in the defined look-up thesaurus term list. If, for instance, 'health, life and disability insurance' is proposed as an expandable pattern, but 'life insurance' is not a known term, then expansion will not happen. If, however, 'health insurance' is not known but 'life insurance' and 'disability insurance' are known, then the (partial) expansion 'health, life insurance and disability insurance' is performed.

The concept expansion must execute after normalization, as Thesaurus term finding usually requires normalized forms to proceed.

6.13.1 Instantiation

```
<!-- if not set, the HeadCoordinations and CommaCoordinations are false ! -->
<a:ExpandConceptCoordinations     x:Name="ECC"
                                Coordinators="Coordinators"
                                HeadCoordinations="true"
                                CommaCoordinations="true"
                                Thesaurus="MeSH2009" />
```

6.13.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="WordIndex" annotationType="TACO.Data.IWord" />
  <Property name="SentenceIndex" annotationType="TACO.Data.ISentence" />
  <Property name="Text" type="System.String"/>
</Input>
<Output>
  <Property name="ExpansionStorage" annotationType="TACO.Data.ArrayExpansion" />
</Output>
```

6.13.3 Example

6.14 Flag Terms

With the FlagTerms Activity you can add flags to terms previously found in the text by the *Find terms* activity. Flags are unary tags associated with a *TermAnnotation*, that are either present or not present. You can add a flag, for instance to mark a term that is must be ignored during creation of the fingerprint (cf. the *Make Fingerprint*) or because you want to do something special with the flagged TermAnnotations during processing of the Elsevier Fingerprint Engine output.

TermAnnotations can be flagged on the basis of two word sets : a set of concept IDs or a set of term IDs. The term is flagged if its ConceptID occurs in the concept set, or if its TermID occurs in the term set (or both, of course). The TermsToFlag and ConceptsToFlag (both optional) attributes must both refer to a *Wordset* static object.

6.14.1 Instantiation

```
<a:FlagTerms        x:Name="FlagTermsByTermID"
                    Flag="ignore" TermsToFlag="IgnoreTerms" />
<a:FlagTerms        x:Name="FlagTermsByConcept"
                    Flag="strange" ConceptsToFlag="StrangeConcepts" />
```

6.15 Annotate Terms

With the AnnotateTerms Activity you can annotate found terms with further, variable flags coming from additional sources, such as semantic type dictionaries, concept or term properties. These flags are added to terms previously found in the text by the *Find terms* activity. The AnnotateTerms activity adds flags that provide further information on a term and is not particularly suitable for marking terms with specific flags that would control the creation of the fingerprint or such. For that type of flagging, use the *FlagTerms* activity.

The AnnotateTerms activity can add attributes to a *TermAnnotation* by looking up its ConceptID and/or TermID. To create lookup tables for each of these the static object types of *ConceptAttributes* and *TermAttributes* are particularly suitable.

The TermAttributes and ConceptAttributes (both optional) attributes must both refer to an *Attributes* static object.

6.15.1 Instantiation

```
<a:AnnotateTerms  x:Name="at" ConceptAttributes="MeSHTreeNumberAttributes" />
<a:AnnotateTerms  x:Name="at" TermAttributes="MeSHPOSAttributes" />
```

6.15.2 Disambiguating terms with AnnotateTerms

When the AnnotateTerms activity is provided with a reference to a *disambiguator*, it can assign flags to specific terms that are found inside a token pattern. For instance, if the term ‘solution’ was found after ‘aqueous’, we may wish to rule it out in a meaning ‘answer to a problem’. A rule like that would read

```
'aqueous' (?'ignore' T/TermID='SolutionTermID'/)
```

In order to execute a rule, a pattern annotator a disambiguator object must be defined that assigns the ‘ignore’ flag to terms based on the (above) matching token pattern. The `IDisambiguator` type currently has two instantiations: the `TermAnnotator` and the `ContextFingerprintDisambiguator` - the latter of which is experimental.

```
<a:AnnotateTerms        x:Name="disambiguate" TermAnnotator="Disambiguate" />
```

6.16 AnnotateHomonyms

The AnnotateHomonyms annotates pieces of text where two terms have been found as ‘Honomym’.

```
<a:AnnotateHomonyms     x:Name="annotateHomonyms" Thesauri="CpxTree" />
```

6.17 Distribute Term Flags

The `DistributeTermFlags` activity may be executed after disambiguation. On the assumption that in general term have a single meaning in a document, we may want to distribute an ‘ignore’ flag that has been assigned to a single term based on its context to all other terms with the same ID in the same document. For instance, suppose we rule out ‘solution’ in a document in the meaning ‘mixture’ - then we may assume that all other occurrences of the same word in the same document should be ignored as well. However, this approach is as good as its underlying assumption: it is of course not at all a matter of necessity that two homonymous words in the same document mean the same thing.

There are two possible ways to limit the scope of this activity.

- This activity's operation may be limited to the terms that originate from the same abbreviation as defined in the text using the `InExpansionsAlone` flag (second line below).
- This activity's operation may be limited to the terms that occur within a certain distance only from the original, triggering term, going by the rationale that different meanings of the same homonym are more likely to occur farther out from each other. For that, set the `MaxDistance` parameter. Distance is measured in terms of the number of tokens.

```
<a:DistributeTermFlags  x:Name="distribute"
                InputFlag="ignore" OutputFlag="ignore" />
<a:DistributeTermFlags   x:Name="distribute"
                InputFlag="ignore" OutputFlag="ignore"
                InExpansionsAlone="True" />
<a:DistributeTermFlags   x:Name="distribute"
                InputFlag="ignore" OutputFlag="ignore"
                MaxDistance="20" />
```

6.18 Find Terms

The Find Terms Activity identifies occurrences of thesaurus terms in the input text. This activity must refer to a *Thesaurus* static object. The thesaurus contains a list of terms; the activity annotates these terms in the text. During term finding, the specific term matching attributes such as case sensitivity, word order sensitivity etc. are checked against the term candidates occurring in the text.

The Find Terms Activity has the following parameters encoded in attributes.

- MaxGapSize : the maximum gap allowed in permutable non-exact terms in this thesaurus.
- FilterMethod : the method to be applied to filtering the terms found in the input text. The following values are legal:
  - None
  - ProperSubset
  - PreferOrdered
  - Restrictive
  - PreferBetter
  - AllowEncapsulation
  - PreferBalanced

  Values can be combined with the | marker
- ThesaurusNameOverride : a string value that, if set, is used to *override* the name assigned to newly added TermAnnotations. By default, Thesaurus-type static objects have their own `Name` property which is copied to the `Thesaurus` property on TermAnnotations, such that the origin of thesaurus terms is always transparent. For particular applications one may choose to override this standard assignment in a specific workflow.

The Key of the *thesaurus* that the FindTerms refers to must be named in the `Thesaurus` attribute; other parameters must be filled out in the as below.

6.18.1 Instantiation

```
<a:FindTerms x:Name="MeSH2009"
                Thesaurus="MeSH2009"
                MaxGapSize="1"
                FilterMethod="Restrictive|PreferBetter" />
```

6.18.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="WordIndex" annotationType="TACO.Data.IWord" />
  <Property name="Sentences" annotationType="TACO.Data.ISentence" />
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="ExpansionIndex" annotationType="TACO.Data.IExpansion" optional="true"/>
  <Property name="POSTagIndex" annotationType="TACO.Data.IPOSTag" optional="true"/>
</Input>
<Output>
  <Property name="TermAnnotations" annotationType="TACO.Data.TermAnnotation" />
</Output>
```

6.18.3 Example

Filtering by Parts-of-Speech

The FindTerms activity has the capability to accept or reject term candidates by specific textual attributes defined for terms, e.g. case sensitivity, spacing, presence of punctuation marks etc. In addition, it is able to perform a check on the Part-of-Speech (POS) tag. That would make it possible for instance to reject the term 'bear' (the animal) in the sentence 'This bears no resemblance ...' because of a POS tag mismatch.

To be able to perform this kind of filtering, both the terms involved must be POS-tagged in the thesaurus, and the text to be processed should be POS-tagged. Tags associated with terms in the thesaurus are stored in a *Term Attribute* static object. The `TermAttribute` object is accessed by the FindTerms activity to check POS tags for terms identified. If the POS tags do not match, the term is rejected. The POS tags for the text at hand must be annotated beforehand by running the *Tag Parts of Speech* activity.

Below is an example of a FindTerms configuration where the Term attribute static object is referenced.

```
<a:FindTerms x:Name="MeSH"  Thesaurus="MeSH"
                              TermPOSTags="MeSHPOSAttributes"
                              MaxGapSize="1"
                              FilterMethod="Restrictive" />
```

Block out terms adapter

In case one performs term finding with two thesauri covering different domains, or two thesauri with on thesaurus (A) taking precedence over the other (B), we can *block out* terms from thesaurus A from the input to the term finding activity referring B. The ins-and-outs of adapters is described in a separate *section*, and a special section describes the usage of the *BlockOutTerms* adapter.

6.18.4 Example

6.19 Hide Patterns

Token patterns that are detected by annotators that yield token patterns can be used in different ways. The Hide Patterns Activity uses TokenPatterns as input and will hide the annotated parts of the input text behind 'empty expansions'. Assuming, e.g. that you have annotated 'Citations' in the text, you can choose to hide these by defining the activity `HidePatterns`.

6.19.1 Instantiation

```
<a:HidePatterns       x:Pattern="Citations" >
```

Some patterns are rather disruptive to the text [for instance, citations] whereas others may not be (e.g. percentages or confidence intervals). Should a pattern be disruptive, it is fair to assume that terms denoting concepts can span beyond these patterns. This can be marked by the 'IsBoundary' flag in the activity.

```
<a:HidePatterns       x:Pattern="Citations" IsBoundary="True" >
```

6.19.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenPatterns" annotationType="TACO.Data.ITokenPattern" />
</Input>
<Output>
  <Property name="Expansions" annotationType="TACO.Data.EmptyExpansion" />
</Output>
```

6.19.3 Example

6.20 Make Fingerprint

The MakeFingerprint activity aggregates the term annotations that have been found to a fingerprint.

```
<a:MakeFingerprint       x:Name="MF" />
```

The MakeFingerprint activity needs `TermAnnotation` annotations as input (the output of the *Find terms* activity) and produces `ConceptAnnotation` annotations.

Should you wish to aggregate term annotations from a certain thesaurus (Find Terms activity) only, then you must apply a `SelectTerms` filtering adapter. The configuration of this adapter is very straightforward. The name of the Thesaurus should correspond to the `Name` property as defined in the static object that contains the Thesaurus source.

6.20.1 Instantiation

```
<a:FindTerms      x:Name="TermFinding" Thesaurus="PlantNames"
                     MaxGapSize="1"  FilterMethod="Restrictive" />
<a:MakeFingerprint  x:Name="MF" Terms="{s:SelectTerms Thesaurus=PlantNames}" />
```

Rank Computation

During the making of a fingerprint, all occurrences of terms pointing to a certain concept are grouped and counted. The term counts are stored in the "AFreq" (Absolute frequency) field for each concept. On top of the frequency count, the ranks are computed. Ranks are represented as floating point numbers in the range between 0 and 1. Currently, two rank computation methods are supported:

- Linear
- SquareRoot

Suppose that term A occurs 5 times, term B 3 times, and C once only. Linear rank computation results in scores 1, 0.6 and 0.2 respectively, expressing the rank of a concept as a proportion relative to the frequency of the most frequent concept. By default, rank computation is linear. SquareRoot rank computation results in the square roots of these figures, smoothing out the biggest gaps between the most frequent 'outlier' concepts and the other concepts that occurred in the document.

```
<a:MakeFingerprint  x:Name="MF" RankComputation="Linear" />
<a:MakeFingerprint  x:Name="MF" RankComputation="SquareRoot" />
```

Excluding specific terms

Under some circumstances you may wish to exclude certain terms from aggregation into the fingerprint. For instance, you may find that some terms are found in contexts where they should not be found. If no specific disambiguation clues are available (for instance, disambiguation by POS tags, see Find Terms activity) or if these disambiguation clues are not restrictive enough, terms can be excluded during fingerprint aggregation. It may also be the case that some concepts are not applicable to the document set that is indexed, or that these concepts have little relevance in the domain. The Term IDs to exclude can be listed in a text file and be added as a *word set* to the runtime configuration. When indexing documents on America, for instance, the concept 'America' may not meaningfully contribute to a document fingerprint as it is redundant. The Concept IDs to exclude can be listed in a text file and be added as a *word set* to the runtime configuration. Configuration may look like below. The terms to ignore then can be ruled out using the *Flag Terms* activity and a flag such as 'Ignore' can be assigned to the term. The MakeFingerprint activity can react to this flag (any flag) and ignore these terms. The `IgnoreFlag` attributes is optional and can be left out; the MakeFingerprint activity then aggregates the term annotations without filtering.

```
<a:MakeFingerprint  x:Name="MF" IgnoreFlag="ignore" />
```

Different concept weights per section

If the structure of a document is known to the Elsevier Fingerprint Engine client, the weighing of concepts in the fingerprint can be made dependent of the document structure. A well-known use case is the distinction between title and abstract. If a concept occurs in the title, we may choose to assign maximal weight to it, while other concepts just need to be balanced against one another. This functionality can be achieved by first communicating the text sections to the Elsevier Fingerprint Engine by assigning Xml sections *during pre-processing*.

In the static objects, we can then define a *concept-weights-per-section object* that assigns the appropriate weights to each section.

```
<a:MakeFingerprint  x:Name="MF" ConceptWeightsDefinition="ConceptWeights" />
```

6.20.2 Separate Fingerprint per section

The MatchFingerprint activity has a special property field called `FingerprintBySection`. By default this field is false and processing is as defined above; setting this flag to `True` leads the processing up a separate execution path.

```
<a:MakeFingerprint  x:Name="MFP"    Terms="{s:SelectTerms Thesaurus=Thesaurus}"
                                    ConceptNameTable="Thesaurus"
                                    FingerprintBySection="True"
                                    IgnoreFlag="ignore" />
```

The result of processing the terms by section is that instead of the (usual) `ConceptAnnotation` output, the activity produces `SectionConceptAnnotation` annotations. By this aggregation methodology, the fingerprint computation is performed section-wise, following the sections that are present in the input. If there are no sections, the method is not executed and MakeFingerprint follows the path it would otherwise follow.

The `SectionConceptAnnotation` annotations are identical from `ConceptAnnotation` but contain an extra field `Section` that signals the section the concept comes from. The Concept IDs are therefore no longer unique per thesaurus per document [as usual] but rather unique per thesaurus per document per section.

The resulting annotations can be selected in the usual way in the TACO DatabaseClient configuration:

```
<AnnotationSave Name="SectionConceptAnnotation"
              Fields="DocID as DocID (BIGINT NOT NULL),
                      Section AS Section (NVARCHAR(100) NOT NULL),
                      ConceptID AS ConceptID (INT NOT NULL),
                      AFreq AS AFreq (INT NOT NULL)"
              DestinationTableName="SectionConcepts"
              Create="true" />
```

Concept Names

There is a distinction between the terms associated with a particular concept and the concept name. Some thesauri contain concepts that are purely for organisational purposes, e.g., to provide a well structured hierarchy. These concepts are not meant to be found in texts, since typically the actual concept is found on a different location in the thesuarus. An example is 'Alcohols', which is a general name for a class of chemical substances, and 'Alcohol' which is a commonly used name for one of these substances in beverages. In this case one would generally associate no terms with 'Alcohols', therefore ensuring that when 'alcohol' is found in a text, the common version of the concept is returned. Other reasons for disconnecting the concept name from the list of terms associated with the concept concept names that are well known so can not be changed but are unfit for indexing, e.g., 'cancer, lung'.

The above distiction also avoids the need for a preferred term.

In the *MakeFingerprint* activity the concept names are returned in the fingerprint. The source where these concept names are retrieved is configured in the activity with the ConceptNameTable attribute. There you can reference your CKEThesaurus and it will return the concept names associated with the concept ids (if specified in the thesaurus).

```
<a:MakeFingerprint  x:Name="MF" ConceptNameTable="ConceptNames" />
```

6.20.3 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Terms" annotationType="TACO.Data.ITermAnnotation" />
  <Property name="Tokens" annotationType="TACO.Data.IToken" />
  <Property name="Sections" annotationType="TACO.Data.ISection" optional="true"/>
</Input>
<Output>
  <Property name="ConceptAnnotations" annotationType="TACO.Data.ConceptAnnotation" />
</Output>
```

6.20.4 Example

6.21 Mark up Words

The MarkupWords activity can flag words in the input with a certain flag. Currently, flags can only be switched to 'true' and not to 'false'. The `IWord` annotation has a member `Flags` that can contain the following values and combinations thereof.

- Depluralized
- StopWord
- Empty

The flag to be set must be specified in the `WordFlag` attribute. The MarkupWords activity needs a reference (by name) to a preconfigured *WordSet* static object.

6.21.1 Instantiation

```
<a:MarkupWords            x:Name="StopWords"  WordFlag="StopWord" WordSetName="StopWords" />
```

In contrast to other activities, the MarkupWords activity does not *add* annotations but *modifies* annotations.

6.21.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="WordIndex" annotationType="TACO.Data.IWord" indexType="TACO.Data.IAnnotationInde
</Input>
<Output/>
```

6.21.3 Example

6.22 Normalize

The Normalize activity computes alternative, normal word forms for input words. Typically, normalization of this type involves removal of inflectional affixes, e.g. *children* -> *child*. Normalization, in many instances, will approximate **stemming**. However, this is not a requirement and stemming is not applied rigorously: e.g. 'normalize' will not normalize to 'normal' but to 'normalize'. Normalization also does more than stemming alone: orthographic differences are also removed, e.g. 'analyse' and 'analyze' will both normalize to 'analyze'.

For many or all non-whitespace, non-punctuation tokens, normalizers add a new `NormedWord` structure that contains the normal form according to this normalizer. Word structures are identified by an identifier; currently this identifier is numerical and implicitly extracted from the normalizer taken from the static objects. `NormedWord` structure contain a score relative to the source: this score is used by e.g. the term finding activity. The relative score factor is expressed in the `Score` parameter. The term finder will prefer non-normalized forms over normalized forms (if the latter score higher).

6.22.1 Instantiation

The Normalize activity must be instantiated with a reference to a *Normalizer static object*.

```
<a:Normalize              x:Name="NormEN"     Normalizer="en" Score=".90" />
```

6.22.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Tokens" annotationType="TACO.Data.IToken" />
  <Property name="Words" annotationType="TACO.Data.IWord" />
</Input>
<Output>
  <Property name="WordStorage" annotationType="TACO.Data.NormedWord" />
  <Property name="MultiTokenWordStorage" annotationType="TACO.Data.MultiTokenWord" />
</Output>
```

6.22.3 Example

6.23 Multi Normalize

The MultiNormalize activity can add alternative word forms for ranges of tokens, e.g. normalize *city(ies)* to *city*. In order to do this it needs a reference to a `PatternNormalizer`. Since this activity is under active development, no further documentation is available at this time.

6.23.1 Instantiation

The MultiNormalize activity must be instantiated with a reference to a *IMultiNormalizer static object*.

```
<a:MultiNormalize              x:Name="NormEN"     MultiNormalizer="enPat" Score=".90" />
```

6.23.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Text" type="System.String"/>
  <Property name="Tokens" annotationType="TACO.Data.IToken" />
</Input>
<Output>
  <Property name="WordStorage" annotationType="TACO.Data.MultiTokenWord" />
</Output>
```

6.24 Tag Parts of Speech

The part-of-speech tagger activity needs a reference to a Language Model static object. Using the training data from the Language Model static objects, the part-of-speech tagger activity can produce POS Tags annotations for the input. Before the part-of-speech tagger can operate, tokens need to be identified. Apart from the key to the POSTagger object, no other configuration is required.

There are two methods of computating POS tags.

- Viterbi : all possibilities are computed in a rather efficient way, but still the Viterbi method is 'brute force' in the sense that *all* possibilities are evaluated
- Beam : an computation method that maintains a 'beam' of a maximum of 5 best possible paths at each position. This engine not only outputs the best Tag for each taggable chunk, but also the next-best alternative (if any) and its distance to the best tag. A larger distance (> 1) indicates a higher confidence level.

By default, the Viterbi method is used. In order to activate the Beam computation, specify the attribute `Beam="true"`.

6.24.1 Instantiation

```
<p:TagPartsOfSpeech       x:Name="TagPOS"     Key="en" Beam="true" />
```

See for more details the separate chapter on *part-of-speech tagging*.

6.24.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="TokenIndex" annotationType="TACO.Data.IToken" />
  <Property name="SentenceIndex" annotationType="TACO.Data.ISentence" />
  <Property name="Text" type="System.String"/>
  <Property name="TokenPatterns" annotationType="TACO.Data.ITokenPattern" optional="true"/>
</Input>
<Output>
  <Property name="POSTags" annotationType="TACO.Data.POSTag" />
</Output>
```

6.24.3 Example

6.25 Tokenize

The Tokenize activity splits the text in Tokens. As an extra, for non-whitespace tokens, base Word forms are also inserted. The Tokenize activity requires no input annotations, but directly processes the input Text. The Tokenize activity must be instantiated with a reference to a *Tokenizer static object*.

6.25.1 Instantiation

```
<a:Tokenize x:Name="tokenize" Tokenizer="SampleTokenizer"  />
```

6.25.2 Input Annotation Interfaces and Output Annotation Types

```
<Input>
  <Property name="Text"      type="System.String"/>
  <Property name="Sections"  annotationType="TACO.Data.ISection"  optional="true"/>
</Input>
<Output>
  <Property name="Tokens"    annotationType="TACO.Data.Token"     />
  <Property name="Words"     annotationType="TACO.Data.Word"      />
  <Property name="Sentences" annotationType="TACO.Data.Sentence"  />
</Output>
```

6.25.3 Example

6.26 Detect Sentences

The DetectSentences activity splits the text and tokens into Sentences. The DetectSentences activity requires only Token annotations. The DetectSentences activity must be instantiated with a reference to a *Sentence Detector static object*.

6.26.1 Instantiation

```
<a:DetectSentences x:Name="dt" SentenceDetector="en"  />
```

6.27 Sample Configurations

Three sample configurations are included in the Elsevier Fingerprint Engine Engine instance installer. It is recommended to install all three of them. Looking at these sample configurations, it is usually possible to customize one to your specific needs.

CHAPTER

SEVEN

WORKFLOW LISTS

The workflow configuration file `WorkflowList.xaml` contains a list of defined workflows. The Elsevier Fingerprint Engine instance embedded in the IIS Server will expose handles for each workflow defined. The URLs for each workflow are based on the default host URL as defined during the Fingerprint Engine *instance installation*.

The server exposes two handles for each workflow: a SOAP interface and a REST interface. The resulting URLs for a workflow as below (SampleWorkFlow) are as follows.

```
REST: BASEURL/WorkFlowName/TacoService.svc : http://localhost/TACO/SampleWorkflow/TacoService.svc
SOAP: BASEURL/WorkFlowName/Xml.svc         : http://localhost/TACO/SampleWorkflow/Xml.svc?wsdl
```

The workflow configuration needs, for each workflow:

- a (unique) name for the workflow,
- a reference to the XOML-file that contains the *configuration of that individual workflow*,
- a reference to the static object that defines the output *annotation filter* for this workflow.
- [optional] a reference (by name) to the *preprocessor* pre-processing the input to this workflow.
- [optional] a reference (by name) to the object that defines an *annotation extender* for the outputs of this workflow.

```
<?xml version="1.0" encoding="utf-8"?>
<Workflows
    xmlns="clr-namespace:TACO.Core;assembly=TACO.Core"
    xmlns:xaml="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <Workflow
        xaml:Key="SampleWorkflow"
        Path="SampleWorkflow.xoml"
        AnnotationFilter="WhiteList"
        />
    <Workflow
        xaml:Key="ExtendedWorkflow"
        Path="SampleWorkflow.xoml"
        AnnotationFilter="WhiteList"
        AnnotationExtender="Extender"
        Preprocessor=XmlPreprocessor"
        />
</Workflows>
```

CHAPTER EIGHT

PREPROCESSORS

Preprocessors are not so much a functional part of the Elsevier Fingerprint Engine: rather, preprocessors filter or adapt the input in order to deliver the correct input to the Fingerprint Engine, that is: text. *Preprocessors* are separate modules that convert and or format a text to make it suitable to submit to the Fingerprint Engine. For instance, a preprocessor could take an HTML document, extract the text to be processed, and add a section annotation to indicate the title, body etc. The *Tokenize activity* for instance, can accept a section annotation and process it.

8.1 NoOpPreprocessor

The no-op preprocessor is the preprocessor that is selected if no other preprocessor has been defined. This preprocessor does nothing. There is no need to specify it; it will be selected by default and does not need to be defined in the static objects.

8.2 XmlSectionsPreprocessor

The XmlSectionsPreprocessor is a conceptually rather simple preprocessor that can handle input that is structured by limited Xml for simple section markup. The Elsevier Fingerprint Engine allows for document segmentation in major units such as *title*, *abstract* and *references*. This segmentation is derived from the input by simply associating `Section` annotations with 2nd level xml nodes in the input text. For example consider the following input:

```
<document>
    <title>this is the title</title>
    <abstract>this is the abstract</abstract>
</document>
```

The XmlSectionsPreprocessor will produce section annotations that designate the first line of text as 'title' and the second line as 'abstract'. These section names can be used in the *fingerprinting activity* to, for example, assign maximum weights to concepts occurring in the title but balanced weights to concepts occurring the abstract section. The XmlSectionsPreprocessor allows for 2-levels of Xml structure only; processing a document like `<xml><first><second></second></first></xml>` will result in an error.

8.3 XmlPreprocessor

The XmlPreprocessor is a conceptually rather simple preprocessor that can handle input that uses full Xml for section markup. The Elsevier Fingerprint Engine only allows for document segmentation in major units such as *title*, *abstract* and *references*. As in the XmlSectionsPreprocessor, this segmentation is derived from the input by simply associating `Section` annotations with 2nd level Xml nodes in the input text.

However, Xml may have more structure or annotations than just two levels of sectioning. Elsevier Fingerprint Engine can also handle input with subscripts or superscripts, or other formatting markup, such as in the example below.

```
<document>
    <title>This is about CO<sub>2</sub>.</title>
    <abstract><emph>Carbondioxide</emph></abstract>
</document>
```

The XmlPreprocessor removes all of the mark-up below the second level. The embedded text between those tags will not be treated as separate sections, but will be tokenized separately. This means that `CO2` is a single token but `CO<sub>2</sub>` are two.

In contrast to the XmlSectionsPreprocessor, the XmlPreprocessor is also very restrictive in Xml parsing, e.g. it will require namespace definitions in order to be able to parse the Xml properly. The definition of an XmlPreprocessor, therefore, may optionally contain namespace definitions defined in the `Collexis.Common` namespace definition format.

```
<st:XmlPreprocessor xaml:Key="fullxml"
    xmlns:st="clr-namespace:TACO.Core.Storage;assembly=TACO.Core"
    xmlns:cc="clr-namespace:Collexis.Common;assembly=Collexis.Common" >
    <st:XmlPreprocessor.Namespaces>
        <cc:Namespaces>
            <cc:Namespace Prefix="ce" URL="http://www.elsevier.com/xml/common/dtd" />
            <cc:Namespace Prefix="ait" URL="http://www.elsevier.com/xml/common/ait/dtd" />
            <cc:Namespace Prefix="sb" URL="http://www.elsevier.com/xml/common/struct-bib/dtd" />
            <cc:Namespace Prefix="xlink" URL="http://www.w3.org/1999/xlink" />
        </cc:Namespaces>
    </st:XmlPreprocessor.Namespaces>
</st:XmlPreprocessor>
```

8.3.1 Section types, metadata

For all Xml sections, the semantics for the section can be defined in the `XmlPreprocessor` configuration. There are a number of possible section types.

- Parent : top level section, containing embedded content sections
- Content : leaf level section, containing actual content. content of this section will be processed as text
- Metadata : metadata section, containing configurational or other metadata that does not get analyzed as text

The `Parent` and `Content` section types cannot be configured but are computed by the preprocessor - determined by the location of the Xml node in the Xml document. In a structured Xml document, the top node is classified as `Parent` whereas embedded nodes are by default `Content`. However, if a section has been explicitly configured as `Metadata`, it will not be put through the text analysis but rather the section is set aside to be used by other processing logic that accesses metadata sections.

Currently, the only class accessing metadata is the *metadata filter* that can be used to control the workflow. Configuration of explicit section assignments is as follows:

```
<st:XmlPreprocessor xaml:Key="fullxml"
    xmlns:st="clr-namespace:TACO.Core.Storage;assembly=TACO.Core"
    xmlns:cc="clr-namespace:Collexis.Common;assembly=Collexis.Common" >

    <SectionType xaml:Key="metadata">Metadata</SectionType>
    <SectionType xaml:Key="Metadata">Metadata</SectionType>

    <st:XmlPreprocessor.Namespaces>
        <cc:Namespaces>
            <cc:Namespace Prefix="ce" URL="http://www.elsevier.com/xml/common/dtd" />
            ...
        </cc:Namespaces>
    </st:XmlPreprocessor.Namespaces>
</st:XmlPreprocessor>
```

The result of this is that from a document such as the following only the `<text />` and `<title />` sections will be processed.

```
<xml>
    <title>this is my document</title>
    <text>this is the text </text>
    <metadata><workflows><workflow>workflow</workflow></workflows></metadata>
</xml>
```

CHAPTER

NINE

STATIC OBJECTS

Runtime configurations contain the configuration of static objects, that are used by activities listed in the workflow configuration.

These objects are functions or datasets needed by the activities. They are referred to in the *activity configuration* by their name, which is looked up in the runtime configuration.

In the runtime configuration these static objects are listed in the following way:

```
<WorkflowRuntime.StaticObjects>
    <NamedObjects>
        <OBJECTTYPE xaml:Key="NAME" OTHERATTRIBUTE1="..." OTHERATTRIBUTE2="..." />
    </NamedObjects>
</WorkflowRuntime.StaticObjects>
```

These named objects have types (OBJECTTYPE) that reflect their implementation in the software. Multiple named objects of the same type (for instance, a *Thesaurus* or *Lexicon*) may be instantiated at the same time but then these objects must be differentiated by name, configured in the attribute `xaml:Key`.

Whatever the type of the objects, what matters to the activities calling on the objects is their interface. For instance, we may instantiate wordsets from a database table or from a file; a *database wordset* will be instantiated from a database table and a *file wordset* may be instantiated from a file, this does not matter to the activity that requires any wordset compliant with the `IWordSet` interface.

The Elsevier Fingerprint Engine runtime detects the relevant interfaces of Static Objects and will group these objects automatically. Static objects are organized *pools* containing objects with the same interface. Static objects are persistent and constant throughout the lifetime of a runtime. Since static objects are organized per interface type; each single static object must have a **Key** that is unique to the static object with that interface type, i.e. in its pool of similar static objects. Other static objects and activities can query the static objects by this **Key**.

We know of the following types of static objects:

- the Normalizer static objects,
- the Tokenizer static objects,
- the SentenceDetector static objects,
- the *Thesaurus static objects*,
- the Attributes static objects,
- the *Language Recognition static objects*,
- the Word Set static object,
- the WordPair static objects,
- the LanguageModel static objects,
- the *Annotation Pattern Annotator static objects*,
- the *Text Pattern Annotator static objects*
- the ConceptWeightsPerSection definitions,

- the OutputFilter objects,
- the AnnotationExtender types.

Below is a table showing which interfaces these static objects support

Type	Interface ILanguageDetector	INormalizer	IOutputFilter	ITokenizer	IWord-Set
Normalizer		X			X
Tokenizer				X	
Thesaurus					X
Language Recognition	X				
Word Set					X
WordPair					
LanguageModel					
OutputFilter			X		
AnnotationExtender					

9.1 Normalizer Static Objects

Normalizer static objects answer to the interface of `INormalizer`. Currently only Elsevier normalizers are supported. Elsevier normalizer objects also implement the `IWordSet` (see *below*) interface, so make sure that you choose a key (name) that is unique within both sets of objects. The WordSet interface of the Elsevier Normalizers represents the entry keys of the normalizer table.

The Elsevier Normalizers are implemented in an assembly for their own language and type. This means that the namespace for each normalizer must be defined uniquely, e.g. as `xmlns:n="clr-namespace:TACO.Normalizers;assembly=TACO.Normalizer.xx"`. This namespace must be declared (as above) before the normalizer can be instantiated.

For that reason, the `TACO.Normalizer.xx` assembly must be present in the `bin\` folder under the Elsevier Fingerprint Engine Service root folder in order for that Normalizer to load.

The following Normalizer types are currently supported:

Type	Contents	Remarks
NormalizerEn	English normalization tables	
NormalizerEnP	English precision normalization tables	
NormalizerDe	German normalization tables	on request – not included in the standard installation
NormalizerDeOO	German normalization tables based on hunspell data	(see above)
NormalizerNl	Dutch normal normalization tables	(see above)
NormalizerNlP	Dutch precision normalization tables	(see above)
NormalizerFr	French normalization tables	(see above)
NormalizerEs	Spanish normalization tables	(see above)
NormalizerPt	Portuguese normalization tables	(see above)

Normalizer static objects contain tables of known normalizations for words of the language specified, and the rules for handling words that are not in the tables. To fill gaps in the tables or override default normalizer behaviour, it is possible to specify a pointer to a user-defined 'overrides' file. The folder where this file is located is `root\config\workflow`. The format of the overrides file is quite specific; it is advisable to consult an expert before defining one.

```
<NamedObjects>
    <n:NormalizerEn xaml:Key="en" Overrides="MeSHExtraNormMappings.txt" />
</NamedObjects>
```

In the above configuration snippet the **Key** is a user-supplied name used to identify the static object. The **NormalizerEn** element denotes which type of static object is referred to here. The **Overrides** attribute refers to the 'overrides' file mentioned above.

9.2 MultiNormalizer Static Objects

Since the multi-token normalization activity is still in beta, documentation is missing at the moment.

9.3 Tokenizer Static Objects

The Tokenizer is a static object that provides the functionality to segment an input text on formal grounds. The tokenizer detects tokens, words and sentences. Tokens are qualified with several flags that indicate the token type.

9.3.1 Tokenizer

The Tokenizer lives in the namespace `xmlns="clr-namespace:TACO.Core;assembly=TACO.Core"` which must be mentioned in the XML pre-amble of the Runtime Configuration.

```
<NamedObjects>
    <Tokenizer xaml:Key="en"
               MaxAbbreviationLength="3"
               LanguageID="en" />
</NamedObjects>
```

The tokenizer requires the following attributes.

- MaxAbbreviationLength: the maximum length of words of Title Case Pattern (Xxx) followed by a dot, to qualify as an abbreviation (with, consequently, the dot not qualifying a sentence separator). If the length is set to 3, for instance, the strings 'X.', 'Xx.' and 'Xxx.' will be counted as abbreviation, but not 'Xxxx'.
- Language: one of "en", "nl", "de", "es", "fr", "pt". The language setting is required to enable language specific escape sequences defined in the normalizer, e.g. to recognize "'s" as a word suffix in English or Dutch.

The following attributes are optional.

- Thoughtful: a boolean flag that can be set to either 'True' or 'False' (default). If set to "true", the tokenizer will first tokenize the text by the standard configuration, and then go through the short words [words that are 1 less in length than the MaxAbbreviationLength], again. In this 'second-strike', the tokenizer will count:
  - the occurrences of free standing short words (type "Xxx"), and then
  - the occurrence of the same words before a dot (type "Xxx.")

  After this count, the frequencies of those words will be used to decide whether the words before the dot [normally interpreted as abbreviations] should actually be re-interpreted as ordinary short words.
- Names: a data source pointing to a list of (short) words that represent names and would be tokenized as abbreviations - if not for being listed here. For example, if the word 'God' is in this list, it will never be regarded as an abbreviation, whereas otherwise, because it is so short, it may be.

```
<NamedObjects>
    <Tokenizer xaml:Key="en" Thoughtful="True" MaxAbbreviationLength="3" LanguageID="en" >
            <Tokenizer.Names>
                <cc:Table>
                    <cc:Thead><cc:Td Value="Name" /></cc:Thead>
                    <cc:Tr><cc:Td Value="God" /></cc:Tr>
                </cc:Table>
            </Tokenizer.Names>
        </Tokenizer>
</NamedObjects>
```

Note: Missing: WordSeparator

9.3.2 PatternTokenizerEn

A specialized tokenizer has been developed for English. The default Elsevier Fingerprint Engine tokenizers contain the logic and knowledge to deal with latin-based languages in general. The PatternTokenizerEn is contained in a separate Assembly (`TACO.Tokenizers`) and has more knowledge of the English language that is used during tokenization. The PatternTokenizerEn has a list of abbreviations and a list of words that frequently occur sentence-initially. This knowledge is used to help distinguishing final periods in dotted abbreviations (e.g., Neurol. or Mass.) from sentence-final periods - these tokens, consequently are not marked as possible sentence boundaries.

Configuration is as follows:

```
<NamedObjects xmlns:t="clr-namespace:TACO.Tokenizers;assembly=TACO.Tokenizers">
    <t:PatternTokenizerEn xaml:Key="en"/>
</NamedObjects>
```

9.4 SentenceDetector Static Objects

The SentenceDetector is a static object that provides the functionality to detect sentence boundaries in a tokenized text. The tokenizer first detects tokens and words and marks the tokens that *could* be a sentence boundary. The sentence detector then sorts out whether there are overriding reasons not to break a sentence at these tokens. Reasons to reject a certain sentence boundary candidate are: breaks inside bracketed areas [that form contiguous units that should not be broken up].

9.4.1 SentenceDetector

The SentenceDetector lives in the namespace `xmlns="clr-namespace:TACO.Core;assembly=TACO.Core"` which must be mentioned in the XML pre-amble of the Runtime Configuration.

```
<NamedObjects>
    <SentenceDetector xaml:Key="en" MaxBracketScope="30" />
</NamedObjects>
```

The SentenceDetector allows for the following attributes.

- MaxBracketScope: the maximum scope of brackets. The SentenceDetector will not find sentence breaks in stretches of text contained within brackets that are shorter than this integer value.
- SentenceBreaking: can have the values `None` (default), `OnNewLine` and `OnBlankLine`. This flag controls the sentence breaking behavior on the occurrence of newlines in the text. If the flag is set to `OnNewLine`, a new sentence will start after every new line [or line feed symbol, or the pair combined]. If the flag is set to `OnBlankLine`, a new sentence will start after every occurrence of multiple new lines / line feeds - in effect, one or more blank lines. *Note* that preservation of newlines in Xml-input is a complicated issue and not guaranteed, cf. http://www.w3.org/TR/REC-xml/.

9.5 Attributes Static Objects

Attribute static objects are used for storing one-to-many associations between keys and values. A single key may be associated with multiple values - the 'attributes'. The following Attributes objects are defined: *TermAttributes* and *ConceptAttributes*.

9.5.1 Term Attributes

A `TermAttributes` object is used for storing term attributes. Term Attributes are attributes of a certain type that are associated with Thesaurus terms - the link is made by the TermID. The attribute must be named, for instance 'POS' as 'Part-of-Speech' or 'LexicalTag'. A `TermAttributes` object is instantiated by providing a link to a *CKE Thesaurus* and naming the term attribute that is desired. During initialisation the attributes will be retrieved from the thesaurus source.

```
<m:TermAttributes Attribute="POS" Thesaurus="MeSH" xaml:Key="MeSHPOSAttributes" />
```

9.5.2 Concept Attributes

A `ConceptAttributes` object is used for storing Concept attributes. Concept Attributes are attributes of a certain type that are associated with Thesaurus Concepts - the link is made by the ConceptID. The attribute is named, for instance as 'TreeNumber' or 'SemanticType'. A `ConceptAttributes` object is instantiated by providing a link to a *CKE Thesaurus* and naming the Concept attribute that is desired. During initialisation the attributes will be retrieved from the thesaurus source.

```
<m:ConceptAttributes Attribute="TreeNumber" Thesaurus="MeSH" xaml:Key="MeSHTreeNumberAttributes"
```

9.6 Language Recognition Static Object

The Language Recognition static object contains a block of statistical data that enable the language recognition activity to determine the language of an input text. The details of the *Language Recognition static object* are described in the document dedicated to *Language Handling*.

9.7 Word Set Static Object

Word sets are simply sets of words. Some activities need sets of words to work with. For instance, the *word-markup activity* can mark up words from a certain set that is statically present in a static object. All WordSet objects have a *boolean* `CaseSensitive` parameter that determines whether the word set data must be treated as case-sensitive or not.

Currently we have the following types of Word Set objects.

- DBWordSet

  A DBWordSet reads a set of words from a single column from a single database table. It needs the following parameters:

  - ConnectionString : ODBC connection string
  - Query : query to retrieve word set data

- SQLWordSet

  A SQLWordSet reads a set of words from a single column from a single database table. The only difference with a DBWordSet is that the connection string is a SQL Connection string rather than an ODBC connection string.

```
<WorkflowRuntime.StaticObjectPools>
    ...
    <!-- InterfaceType="IWordSet" -->
    <NamedObjects>
        <s:DBWordSet xaml:Key="StopWords"
            ConnectionString="DRIVER={MySQL ODBC 5.1 Driver};SERVER=localhost;
                        DATABASE=test;OPTION=3;USER=#;PASSWORD=*;"
            Query="select test from test"
            CaseSensitive="false" />
    </NamedObjects>
    ...
</WorkflowRuntime.StaticObjectPools>
```

- FileWordSet

  A FileWordSet is a word set that takes its data from a file (one word per line) that is pointed to by its location on the file system. This location may either be presented by an absolute path or by a path relative to the root path of the Elsevier Fingerprint Engine instance configuration.

```
<!-- InterfaceType="IWordSet" -->
<NamedObjects>
    <s:FileWordSet xaml:Key="StopWords"  FileName="MeSHStopWords.txt" CaseSensitive="false" />
</NamedObjects>
```

9.8 WordPair Static Objects

WordPair static objects contain a table that is accessible by two words: word A and B. WordPair static objects provide an easy way to determine if two words maintain a certain relationship and function as a two-dimensional dictionary with an easy lookup for table[A][B]. The only input for a word pair static object is a two-column table with wordpair A,B on each line. If the `Ordered` flag is false, both table[A][B] as well as table[B][A] are found. The `IWordPair` interface also provides access to the the table by allowing for a prefix search on both A and B: given A, an `IWordPair` compliant static object can determine if a prefix B can be found for a following word, leaving the `remainder` number of characters left on the word searched. This rather peculiar piece of functionality is very useful for finding coordinations in the input text, where this use case occurs to establish the coordination validity of e.g. “two- and threefold”, where, obviously, the wordpair “two” and “three” are coordinated. WordPair static objects are therefore required with the *coordination expanding* activity. The only current implementation of the WordPair static object is from a text file.

```
<NamedObjects>
    <s:FilePrefixPair   xaml:Key="Coordinations"
                        FilePath="Coordinations.txt"
                        Ordered="false"
                        Remainder="3" />
</NamedObjects>
```

9.9 Abbreviation Settings

The Abbreviation Settings static object controls the settings for abbreviation detection. Currently this static object can only be configured for language.

The Abbreviation Settings have been introduced as a static object because of the large number of detail settings that ensue from a single language code. Abbreviation definitions are highly culture-specific and the likelihood of certain contexts and certain ‘trigger’ words are language-specific. For instance, it is known for English that ‘e.g.’ introduces an example [thus not an abbreviation] and that ‘and’ is an insignificant word. Similar knowledge is stored for all languages - thus the language setting control abbreviation detection. Abbreviation Settings static object is required with the *abbreviation detecting* activity.

```
<NamedObjects
    xmlns="clr-namespace:TACO.Core;assembly=TACO.Core"
    xmlns:abb="clr-namespace:TACO.Core.AbbreviationHandling;assembly=TACO.Core" >
    <abb:AbbreviationSettings LanguageID="en" xaml:Key="AbbrevEn" />
</NamedObjects>
```

9.10 Concept Weights Per Section

In routine situations a document often consists of various sections. When creating a fingerprint using the *MakeFingerprint* activity you may want to aggregate the terms found in these sections differently for each section. This can be achieved using the `SectionWeights` static object. Using this object one can define for named sections how the terms found in that section should be weighted in the final fingerprint. There are three types of SectionWeight:

- `ForcedRankSection`: All terms found in this section will have a fixed rank equal to that specified by the `Rank` attribute (a double value). If multiple `ForcedRank` sections are present, the one with the highest rank takes precedence.
- `WeightedSection`: The `Weight` attribute (a double value) determines the relative weight of this section during combination and aggregation of the different sections into a single fingerprint.
- `IgnoredSection`: This section is ignored during combination and aggregation of the different sections into a single fingerprint.

In all cases, the `xaml:Key` attribute designates the section that the SectionWeight applies to. If the *MakeFingerprint* activity finds a section in the input that is not defined in the defined SectionWeights object, it will throw an exception.

The example below will cause all concepts found in the title to get Rank 1, while weighing the ‘abstracts’ and ‘text’ sections equally and ignoring concepts found in the ‘methods’ section.

```
<NamedObjects>
  <s:SectionWeights xaml:Key="weights">
    <ForcedRankSection Rank="1"   xaml:Key="title" />
    <WeightedSection   Weight="1" xaml:Key="abstract" />
    <WeightedSection   Weight="1" xaml:Key="text" />
    <IgnoredSection               xaml:Key="methods" />
  </s:SectionWeights>
</NamedObjects>
```

9.11 LanguageModel Static Objects

A standard Elsevier Fingerprint Engine installation contains data for English-language POS tagger static objects. These objects are Hidden Markov Models that are executed by a tagging engine. The tagging engines require a ‘language model’ - a representation of the rules for the language. The models are stored in compiled format in the assemblies.

The data in the Language Models can be referenced by the *Part-of-speech tagging activity* that contains the logic for executing the language model. The POS-tagger activity yields tagged token range annotations that are of the type `POSTag`. `POSTag` annotations contain a member property `Tag` that has a value that is specific to the POSTagger at hand. The definition and contents of the tag sets vary according to the tagger and are described in documentation specific to the POSTagger.

The best performing POSTagger in Elsevier Fingerprint Engine is the “en” language model which was trained on the Brown corpus and yields a precision of 98% on the test set, and more than 96% on unseen data. Configuration is as follows.

```
<!-- InterfaceType="ILanguageModel" -->
<NamedObjects>
```

```
    <p:HiddenMarkovModel LanguageID="en" xaml:Key="en" />
</NamedObjects>
```

9.12 OutputFilter Static Objects

An OutputFilter removes annotations from the XML serialization of annotations.

To use an OutputFilter in a workflow, the following is needed:

- the output filter must be defined in RuntimeConfiguration.xaml
- the workflow must refer to the output filter in WorkflowList.xaml

There are two output filter classes:

- a `BlackListAnnotationFilter` class that accepts all annotations except those that are of one of the types in a list of types, and
- a `WhiteListAnnotationFilter` class that accepts only annotations that are of one of the types in a list of types.

If you want to have a workflow to return all generated annotations, use a BlackListAnnotationFilter without further settings: since no annotations have then been blacklisted, all will be returned. Output size can then be further trimmed down by naming the annotation types that you *don't* want. A WhiteListAnnotationFilter works the other way around: no annotations are returned *unless* specified.

By default, the input text is *not* returned by the Elsevier Fingerprint Engine. Should you want the input text returned, one must set the “SerializeText” property to “True” on annotation filter - either BlackList or WhiteList.

An example:

9.12.1 RuntimeConfiguration.xaml:

```
<!-- InterfaceType="IOutputFilter" -->
<NamedObjects>
  <!-- an output filter that accepts all annotations, regardless of their type: -->
  <d:BlackListAnnotationFilter xaml:Key="DefaultAnnotationFilter"/>

  <!-- an output filter that sends back all annotations, including the input text: -->
  <d:BlackListAnnotationFilter xaml:Key="DefaultText" SerializeText="True" />"

  <!-- an output filter that accepts only annotations of type TACO.Data.ConceptAnnotation: -->
  <d:WhiteListAnnotationFilter xaml:Key="KeepOnlyConceptAnnotations">
    <d:WhiteListAnnotationFilter.Types>
      <d:Types>
        <d:Type Name="TACO.Data.ConceptAnnotation" />
      </d:Types>
    </d:WhiteListAnnotationFilter.Types>
  </d:WhiteListAnnotationFilter>
</NamedObjects>
```

9.12.2 WorkflowList.xaml:

```
<Workflows>
  <!-- a workflow that uses the 'KeepOnlyConceptAnnotations' annotation filter: -->
  <Workflow
      xaml:Key="SampleWorkflow"
      Path="SampleWorkflow.xoml"
      AnnotationFilter="KeepOnlyConceptAnnotations"
  />
</Workflows>
```

9.13 AnnotationExtender Static Objects

Annotation extenders add fields to or change fields in the XML serialization of annotations. Examples include:

- Add TextOffset and TextOffset fields to an annotation that specify the range of input characters the annotation is made on (thus making it easier for a client to visualize the annotated text)
- Add a ‘Text’ field that contain the text of an annotation that refers to a range of tokens

To use an extender in a workflow, the following is needed:

- the extender must be defined in RuntimeConfiguration.xaml
- the workflow must refer to the extender in WorkflowList.xaml

A workflow can ony use at most one extender, but a extender can specify multiple kinds of expansions. For example:

9.13.1 RuntimeConfiguration.xaml:

```
<!-- an AnnotationExtender that extends three types of annotations -->
<d:AnnotationExtender xaml:Key="SampleAnnotationExtender">
  <d:AnnotationExtender.Extenders>
    <d:Extenders>
      <d:Extend Type="TACO.Data.Token" Enable="ITextRange"/>
      <d:Extend Type="TACO.Data.Word" Enable="Word"/>
      <d:Extend Type="TACO.Data.Abbreviation" Enable="IAbbreviation"/>
    </d:Extenders>
  </d:AnnotationExtender.Extenders>
</d:AnnotationExtender>
```

9.13.2 WorkflowList.xaml:

```
<Workflows>
  <!-- a workflow that uses the 'SampleAnnotationExtender' Annotation Extender -->
  <Workflow
      xaml:Key="SampleWorkflow"
      Path="SampleWorkflow.xoml"
      AnnotationExtender="SampleAnnotationExtender"
  />
</Workflows>
```

CHAPTER

TEN

THESAURUS STATIC OBJECTS

Thesaurus static objects represent Elsevier Fingerprint Engine thesauri. Elsevier Fingerprint Engine thesauri are automated thesauri that can be used to find terms present in the thesaurus in the input text.

Each Thesaurus must be configured for language, in order to select the terms of the right language from the thesaurus. A thesaurus has a, preferably unique, `Name` attribute that contains the name that this thesaurus will give to the Term and Concept annotations produced by the term / concept finding procedures contained in this thesaurus.

To perform initialization, the thesaurus needs a reference to

- a Terms-and-Concepts data source,
- a Normalizer static object,
- a Tokenizer static object

During initialization of the thesaurus, all terms are read from the datasource and pre-processed into term and concept indexes. During this preprocessing step, the normalizer and tokenizer are deployed as analytic tools. These tool must match the normalizer and tokenizer that are later deployed in the Text analysis workflow. Should there be a mismatch, terms may not be found or the results may be otherwise be unpredictable.

10.1 Settings; Features

10.1.1 Stopwords

For some purposes, the user may choose to ignore so-called stopwords: linguistic function-words that are very frequent and relatively meaningless respective to the purpose of the conceptual indexation of the text. The indexer can ignore these words and must also leave them out of the term indexes built during thesaurus initialization. There are two ways of defining stopwords:

- Referencing a stopwords static object (of the *IWordSet* interface) by an attribute `StopWords` on the Thesaurus Xaml-definition.
- Referencing the key for stopwords in the *CKE thesaurus* `SpecialWords` table. This table contains special-purpose words, one type of which is the stopword. Normally configuration of this would look like * StopWordKey="StopWord". The thesaurus object then loads the stopwords from the same Elsevier Fingerprint Engine datasource as the other thesaurus data.

These references are made by the respective keys of the normalizer and tokenizer static objects. In the Xaml-configuration, these references are made by attributes on the Thesaurus Xml node. It is required that the normalizer and tokenizer static objects are defined prior to the definitions of the thesaurus object.

10.1.2 Case insensitive searching

As an additional configurational feature, the thesaurus can be made to search for lowercase forms of all-capital words by add an attribute `SearchCaseInsensitive="true"`. Even if this flag is set, words with matching capitalisation will still be preferred.

10.2 Defining Thesaurus static objects

The Thesaurus static objects are defined in the namespace `xmlns:m="clr-namespace:TACO.Modules;assembly=TACO.` Elsevier thesaurus objects also implement the `IWordSet` (see *below*) interface, so make sure that you choose a key (name) that is unique within both sets of objects. The `IWordSet` interface of the Elsevier Thesauri represents the entries of the thesaurus single-token search tables.

There are currently three supported database implementations of the CKE thesaurus:

- *SQLite*
- *Oracle*
- *MS SQL Server*

10.2.1 CKEThesaurus

The preferred and simplest way to configure a thesaurus is the Elsevier Fingerprint Engine Thesaurus, written as *CKEThesaurus*. Many thesauri (MeSH, NAL, et al.) can be delivered by Elsevier in this format. A translation tool from SKOS or other Xml formats can be run on request.

Elsevier Fingerprint Engine Thesaurus databases may contain one or more concept graphs, enabling the storage of multiple views on the same thesaurus data, or even the storage of multiple thesauri in a single database. Definition of the GraphID [always a single letter] is optional, because there is also a default concept graph. Similarly, definition of the Language is optional, because there is always a default language.

Configuration of a CKE thesaurus is relatively easy and best explained by example.

```
<NamedObjects xmlns:sq="clr-namespace:SQLThesaurus;assembly=SQLThesaurus">
    <!-- example of a CKE Server thesaurus -->
    <sq:SQLCKEThesaurus ConnectionString="Server=(local);Database=Math2012;Trusted_Connection=Yes
                 xaml:Key="Math"
                 GraphID="M"
                 Normalizer="enp"
                 Language="en"
                 Tokenizer="en"
                 Name="Math" />
</NamedObjects>
```

The nice thing about using the standard Elsevier Fingerprint Engine thesaurus is that the Elsevier Fingerprint Engine takes care of the internal structuring of the database and the handling of the data. Should you use another type of database, the user will need extensive consultancy on the required tables and fields as well as on other aspects of the thesaurus data.

A Elsevier Fingerprint Engine Thesaurus can be prepared for *querying* by setting the `Queryable` attribute to true.

```
Queryable="True"
```

10.2.2 Incremental (composite) term finding

Thesauri can be configured to find terms in an incremental [composing] way, using term-fragments to combine into larger terms. For instance, the chemical *Dextromethorphan Hydrobromide, (+-)-Isomer* may be composed of two fragments: *Dextromethorphan Hydrobromide* and *(+-)-Isomer*. Each of these term parts may have its own matching characteristics, for instance, the *(+-)-Isomer* may be split again to represent the fact that the symbols +- are required whereas other symbols *()* are not distinctive.

The term parts that can be matched separately are called *Fragments*. In order to find terms by *composing* the terms from independently found fragments, the thesaurus must be configured to read Fragments first from a separate graph, and then read the mapping from Fragments of the main graph to Terms on the subsidiary (Fragment) graph from the FragmentAttribute table. The [Key] of the attributes that define this mapping must therefore also be defined.

```
FragmentGraph="c"
FragmentMappingToTerms="Separated"
```

The incremental, composite finding of terms is a highly specialized but very powerful and performance-enhancing feature of the Elsevier Fingerprint Engine. To get a good understanding of the functionality, refer to the relevant special documentation.

TimeOut

It is theoretically possible that a time-out occurs in the connection to the Elsevier Fingerprint Engine thesaurus database. The connection string can be adapted to avoid that problem, see Microsoft help. If that does not solve the problem, consider the possibilities that

- the database may not actually exist or
- the database may not be accessible by the user trying to access it.

10.2.3 OracleCKEThesaurus and SQLiteCKEThesaurus

The Oracle Elsevier Fingerprint Engine Thesaurus static object `OracleCKEThesaurus` parallels the 'default' CKEThesaurus but connects to an Oracle-hosted thesaurus database.

The SQLite Elsevier Fingerprint Engine Thesaurus static object `SQLiteCKEThesaurus` parallels the 'default' CKEThesaurus but reads data from a SQLite database on disk.

10.2.4 GenericThesaurus

If no standard CKE / SQL Server thesaurus can be featured on the TACO host machine, an alternative is the `GenericThesaurus`. This type of thesaurus loads data from an arbitrary [Odbc / Ole / Sql / Xml / Csv] data source. An `IDataSource` object must be assigned to the GenericThesaurus.DataSource handle, such that the thesaurus can read data from the data source during initialization. This process may take some time. There is a variety of implementations for the `IDataSource` interface (see documentation on *Collexis.Common* for this). Here we take DatabaseDataSource as the example. The DatabaseDataSource needs three parameters:

- `DBType` : one of 'Sql', 'Ole' or 'Odbc'.
- `ConnectionString` : the connection string to the data source. The formatting of the connection string is dependent of the connection type and documentation of these connection strings can be found elsewhere.
- `Query` : the query to be executed on the connection.

The GenericThesaurus can read data from a Thesaurus Management Database (TMDB) as we know it from the Collexis version 6 product line. The GenericThesaurus expects a pointer to a SQL connection string that enables it to read the term table from a TMDB. [Note that a Sql ConnectionString can either refer to a named or unnamed instance of a SQL server, e.g. `SERVER=MYSERVER` or `SERVER=MYSERVER\SQLEXPRESS` - which is appropriate depends on your server configuration.]

The SQL query must yield the appropriate fields from the TMDB term table. The required and optional fields that the Query must yield are discussed *below*.

```
<NamedObjects
        xmlns:cc="clr-namespace:Collexis.Common;assembly=TACO.Core"
        xmlns:m="clr-namespace:TACO.Modules;assembly=TACO.Core">

    <!-- example of a Sql Server thesaurus -->
    <m:GenericThesaurus xaml:Key="MeSH"
                 Normalizer="en"
                 Language="en"
```

```
                    Tokenizer="en"
                    StopWords="StopWords"
                    Name="MeSH2011" >
        <m:GenericThesaurus.DataSource>
            <cc:DatabaseDataSource
                ConnectionString="Server=(local);Database=mesh2011;Trusted_Connection=Yes;"
                Query="SELECT * FROM Term"
                DBType="Sql"
                />
        </m:GenericThesaurus.DataSource>
    </m:GenericThesaurus>

    <!-- example of an Odbc thesaurus -->
    <m:GenericThesaurus xaml:Key="MeSH2011"
                    Normalizer="en"
                    Language="en"
                    Tokenizer="en"
                    Name="MeSH2011" />
        <m:GenericThesaurus.DataSource>
            <cc:DatabaseDataSource
                ConnectionString="DRIVER={MySQL ODBC 5.1 Driver};SERVER=localhost;
                                       DATABASE=mesh2012;OPTION=3;USER=#;PASSWORD=*;"
                Query="SELECT * FROM Term"
                DBType="Odbc"
                />
        </m:GenericThesaurus.DataSource>
    </m:GenericThesaurus>
</NamedObjects>
```

Other configuration parameters for Term Finding are defined per workflow, as on the *FindTerms activity*.

For further documentation on the functionality of thesauri, see the functional documentation.

10.2.5 What we need to know about terms

The above configuration of the Thesaurus object shows a query string that produces the list of terms that this thesaurus object handles. These terms are stored in an internal datastructure. This internal datastructure, thenceforth, is the Static Object that answers to the IThesaurus interface. IThesaurus-interfaced objects are used for lookup during processing of documents.

The terms must come from a query. This query must yield at the least the following required field:

- Term : term (string)

All other fields are optional. If these fields are not listed, they default to either an enumeration (for the `Id` and `ConceptId` fields) or to default values (for all other fields).

- TermID : term id (string)
- ConceptID : concept id (string)
- LanguageID : language id (string, 2-letter ISO-639 code; default 'en')
- Permutable : determines word order sensitive of the term (integer, either 1 or 0; default 1)
- Normalizable : determines whether words in the term will be found normalized or not (integer, either 1 or 0; default 1)
- Exact : flag that determines whether punctuation in the term is taken into account (integer, either 1 or 0; default 0)
- CaseSensitive : flag that determines whether lookup is done case-sensitively (integer, either 1 or 0; default !normalizable)

- IgnoreStopwords : determines whether stopwords are allowed in this specific term, i.e. if 'of' and 'the' are stopwords, 'lung cancer' and 'cancer of the lung' will be found if stopwords are allowed in this term. (integer, either 1 or 0; default 1)
- RequireSpacingLeft : determines whether the term must be at a whitespace boundary at the left hand side (integer, either 1 or 0; default 0)
- RequireSpacingRight : same as RequireSpacingLeft at the right hand side

Elsevier uses the TMDB (Thesaurus Manager Databases) that has a standard structure. The 'Term' table of these TMDBs has an appropriate structure to yield the information required for a Thesaurus object. (The fields for **case sensitive** and **allow stopwords** are currently missing, however.) The default query, then, is:

```
SELECT * FROM Terms
```

In MeSH thesauri containing UMLS ids in the ConceptAttribute table, these can be returned in the query by a join.

```
SELECT Id, ca.AttributeValue, Term, LanguageId, Permutable,
        Normalizable, Exact, CaseSensitive, IgnoreStopwords
    from Terms t join ConceptAttributes ca on t.ConceptId = ca.ConceptId
        where ca.AttributeName = 'UMLS'
```

CHAPTER

ELEVEN

ANNOTATION TYPES

The following annotation types are currently available for use in activities. Note that for TokenRanges there are a number of subtypes.

- *TACO.Data.Abbreviation*
- *TACO.Data.ConceptAnnotation*
- *TACO.Data.LanguageAnnotation*
- *TACO.Data.TermAnnotation*
- *TACO.Data.Token*
- *TACO.Data.TokenRange*
- *Expansion types* that give alternative readings for a text, adding or deleting ranges of tokens
  - `TACO.Data.Expansion`
  - `TACO.Data.ArrayExpansion`
  - `TACO.Data.EmptyExpansion`
- *TokenPattern types* that associate pattern information to a range of tokens
  - `TACO.Data.TokenPattern`
  - `TACO.Data.MemberedTokenPattern`
  - `TACO.Modules.FixedStringAnnotation`
- *Word types* giving textual content alternatives and features for (ranges of) tokens
  - `TACO.Data.Word`
  - `TACO.Data.MultiTokenWord`
- *TACO.Data.POSTag*
- TACO.Data.Sentence
- TACO.Data.AllCapsRegion

11.1 Abbreviation

Abbreviation annotations have a list member of attestations, i.e. occurrences of the short-form / long-form pairs in the text. Each attestation consists of a short-form and a long-form, each of which is a TokenRange. Below is the result of processing the input text `Malaria Falciparum (MF) is a disease in Africa, transmitted by mosquitos. MF is common:`

```
<Annotation i:type="Abbreviation">
    <End>6</End>
    <Offset>5</Offset>
    <LongForm>
```

```
        <End>3</End>
        <Offset>0</Offset>
    </LongForm>
    <ShortFormTokens>
        <a:string>MF</a:string>
    </ShortFormTokens>
</Annotation>
```

In the example the long form is spanning tokens 0, 1, 2 (`Malaria Falciparum`), while the short format is spanning token 5 (`MF`). Note that the space and the opening parenthesis are separate tokens 3 and 4.

11.2 Language Annotation

The language annotation is produced only once per document and contains the language of the text at hand. The language information is encoded as a two-letter ISO-639-3 code, e.g. `en` for English or `pt` for Portuguese. The language annotation is produced by a *Detect language* activity.

```
<Annotation i:type="LanguageAnnotation">
            <Language>en</Language>
</Annotation>
```

11.3 Term Annotations

Term annotations are annotations that denote that several stretches of text together may represent a term taken from a controlled vocabulary [such as a Thesaurus]. Term Annotations are not properly token ranges, since terms can be recognized in a non-contiguous (meaning: containing gaps) number of tokens. For that reason, the tokens that are part of the term are listed as an array. For each term, the concept and term IDs are presented as properties. Other fields of a term annotation include:

- OrderingQuality: an assessment of the quality of ordering, i.e. whether the order of elements approximates the order of elements in the thesaurus;
- MatchScore: an assessment of the quality of the term match. For exact terms, this match score is 1, for normalized terms, the score is a little less. This score is used to evaluate competing terms against one another: exact terms out-score normalized terms. The match-score should not be used for ranking terms against one another.
- Flags

11.3.1 Flags

The 'Flags' field may contain a variable number of flags. Some flags are set during *Term finding*, such as the value 'fraction'. Other flags are completely variable and can be set after term finding by term-post-processing activities such as *FlagTerms* and *Annotate Terms*.

The Flag value 'fraction' denotes a term annotation that was based on a part of a word: part of the term does not span a full word. For instance, "HUVEC" may not be a thesaurus entry term but be defined in the text as an abbreviation, short for "Human Umbilical Vein Endothelial Cells". Now, the thesaurus may contain entries for "Umbilical Vein" [UV] and "Endothelial Cells" [EC]. Wherever the abbreviation occurs in the text, both terms (UV and EC) will be annotated - but as fractions. If there were an entry term 'Human Umbilical Vein Endothelial Cells' in thesaurus and 'HUVEC' were annotated like that, it would *not* be a fraction as the entire short form is covered by the term. A TermAnnotation spanning a partial abbreviation *and* a full word is still a fraction, just as 'one-and-a-half' is as much a fraction as 'a half'.

```
<Annotation i:type="TermAnnotation">
    <ConceptID>447435</ConceptID>
    <Flags />
    <MatchScore>0.88</MatchScore>
    <OrderingQuality>1</OrderingQuality>
    <TermID>7437</TermID>
    <Thesaurus>mesh2009</Thesaurus>
    <Tokens>
        <d2p1:int>0</d2p1:int>
        <d2p1:int>2</d2p1:int>
    </Tokens>
</Annotation>
```

11.4 Concept Annotations

Concept annotations are aggregated *term annotations*. The text position of terms cannot be determined from the concept annotations; only the ranking of concept based on the entire document is represented. For both concept and term annotations, the name of the thesaurus containing the terms is represented. This makes it possible to reference multiple thesauri in one Fingerprint Engine call and distinguish ensuing annotations later on.

```
<Annotation i:type="ConceptAnnotation">
    <AFreq>1</AFreq>
    <ConceptID>447435</ConceptID>
    <Rank>1</Rank>
    <Thesaurus>mesh2009</Thesaurus>
</Annotation>
```

11.5 Token

The `Token` type is to a large extent basic for all Fingerprint Engine analyses. Most workflows typically start with *tokenization*. Most Fingerprint Engine analysis activities need Tokens as input; the text tokens are the basic unit for analysis. For each token, the following properties are defined:

- Offset, End : range in the text that the token describes. Note that both are zero-based, and are expressed in number of characters. Also note that `End` is referring to location of the first character *after* the token.
- Capitalisation : capitalisation pattern of the token. This property may have the values None, InitialCap, MiddleCap and Lowers. These values are flags and can combine. For instance, a token without Lowers but not None is all capitalised.
- Type: the token type. This property is also a set of flags that can combine. At least the main type is marked, one of: Alpha, White, Numeric, Alphanumeric, Mixed (alphanumeric with dots) and Punct. Other additional values are: `HasWhitespaceLeft` and `HasWhitespaceRight`, `NoWhitespaceLeft`, `NoWhitespaceRight` indicating what is preceding and following the token.

Below is an excerpt of the output annotation of a Tokenizer supplied with the input text `This is a sample text`

```
<Annotation i:type="Token">
    <Capitalisation>InitialCap Lowers</Capitalisation>
    <End>4</End>
    <Offset>0</Offset>
    <Text>This</Text>
    <Type>Alpha HasWhitespaceLeft HasWhitespaceRight</Type>
</Annotation>
<Annotation i:type="Token">
    <Capitalisation>None</Capitalisation>
    <End>5</End>
    <Offset>4</Offset>
    <Text/>
    <Type>White NoWhitespaceLeft NoWhitespaceRight</Type>
</Annotation>
```

The first two token annotations are shown. The first token is the word `This`, which starts at character 0 (`T`) and ends before character 4 (`space`). Its capitalization is a first capital character and then lowercase. If the text would have been `ThiS` then capitalization would have been characterized by `Mixed`. The token type is characterized as Alpha, surrounded by whitespace on both sides. Note that the start of a text is also characterized as whitespace.

11.6 Token Range

A token range is, obviously, specifying a range of tokens. Many annotation types are based on tokens, particularly the annotations that refer to a specific range of text. The properties for each type are different; however, some properties are shared between different types. The detected tokens are stored as an array, and most range annotations are based on these tokens. To reflect the fact that these annotations refer to tokens, these annotations define their position in the text in terms of positions in the token array. These annotations are, in other words, token ranges. The following annotations are token ranges:

- TACO.Data.AllCapsRegion
- TACO.Data.ArrayExpansion
- TACO.Data.Expansion
- TACO.Data.EmptyExpansion
- TACO.Data.Word
- TACO.Data.MultiTokenWord
- TACO.Data.POSTag
- TACO.Data.Sentence
- TACO.Data.TokenPattern
- TACO.Data.TokenRange
- TACO.Modules.FixedExpression
- TACO.Modules.FixedStringAnnotation
- TACO.Modules.MemberedTokenPattern

All token ranges have a number of properties in common. These will be described here, while properties specific to one or more of the types listed above will be described in the corresponding section. The properties `Offset` and `End` are common to all token ranges. They designate the offset and end of the token range in the underlying array of tokens. Note that these properties are also zero-based and are expressed in number of tokens in contrast to the `Token` annotation, that uses number of characters. An example token range annotation is shown below, as output of tokenizing the text `This is a sample text`

```
<Annotation i:type="Sentence">
    <End>9</End>
    <Offset>0</Offset>
</Annotation>
```

This example is a token range of the `Sentence` type. Note that the entire text contains 9 tokens: 5 words and 4 interword spaces. The `Sentence` token range starts at token 0 (`This`) and end at the nonexisting token 9.

11.7 Word

The tokenizer annotates a Word annotation for all tokens (or token ranges) that are a meaningful alphanumeric unit. Subsequent activities (such as *normalize*) can assign normal or variant forms for these words, stored in Word annotations. The `Offset` and `End` properties are converted in this annotation type to the `TextOffset` and `TextEnd` properties by an *annotation extender*. Word annotations have the following properties:

- Flags* : flags associated with the word (see *the mark-up activity*)

- Score* : this value is a confidence value for the annotation; for Words it is always 1
- Text* : the textual content of this particular word form.
- TextOffset : the location of the start of the word expressed in number of characters
- TextEnd* : the location of the end of the word expressed in number of characters. Note that this value refers to the first character *not* part of the word.

Note that a Word spans exactly one `Token`. If a Word spans multiple tokens (which can be the case for reconstructed words as a result of *de-hyphenation*) the word is stored as a `MultiTokenWord`.

```
<Annotation i:type="Word">
    <Flags>Empty</Flags>
    <Score>1</Score>
    <Text>sample</Text>
    <TextEnd>16</TextEnd>
    <TextOffset>10</TextOffset>
    <Token>6</Token>
</Annotation>
```

The example annotation above is from the input text `This is a sample text`. Note that the `Word` annotation refers to token 6, which spans the characters starting at 10 ending just before character 16.

11.8 Expansions

Expansions have a scope designated by the `Offset` and `End` properties common to all token ranges. A variant of the Expansion type is the EmptyExpansion. This is typically an expansion that is produced at the point in the text where the long-form short form combination is detected. There the short form is removed during expansion, since semantically the author does not mention the term twice.

The expanded form of this token range is denoted by an array property named `Alternative`, that designates the token range that can replace the original input. Below is the result of processing the input text `Malaria Falciparum (MF) is a disease in Africa, transmitted by mosquitos. MF is common`. As described in the *Abbreviation* section, the abbreviation `MF` is connected to the long form `Malaria Falciparum`.

```
<Annotation i:type="EmptyExpansion">
    <End>7</End>
    <Offset>4</Offset>
</Annotation>
<Annotation i:type="Expansion">
    <End>27</End>
    <Offset>26</Offset>
    <Alternative>
        <End>3</End>
        <Offset>0</Offset>
    </Alternative>
</Annotation>
```

The first expansion annotation is an EmptyExpansion, expanding tokens 4 to 6 (text `(MF)`) to nothing. The second expansion annotation is a regular Expansion, expanding token 26 (text `MF` in the second sentence) with tokens 0 to 2.

11.9 POSTag Annotation

POSTag annotations are token ranges with a `Tag` property designating the part-of-speech label assigned to this input text. If the `BeamEngine` POSTag computation engine is used (see the *POSTagger activity*), the nearest alternative tag for this tag is also annotated in a property `Alternative`, as well as its proximity on a logarithmical scale. The proximity is expressed as `Distance`; if this figure is e.g. 5, this means that the alternative tag is 10^-5 as likely as the first choice. For English, a subset of the Brown tagset has been chosen as possible tags.

Below is a POS tag annotation of the text `This is a sample text`:

```
<Annotation i:type="POSTag">
    <End>7</End>
    <Offset>6</Offset>
    <Alternative>vb</Alternative>
    <Distance>0.03375201829564034</Distance>
    <LexicalSource>0</LexicalSource>
    <SemTag>nn</SemTag>
    <Tag>nn</Tag>
</Annotation>
```

This annotation refers to token number 6 (`sample`), which is tagged as a `nn`: a noun. Note that an alternative tag is `vb` (verb), but that is 10^-0.034 = 0.92, so almost as likely.

The list of possible POS tags is large, and is described in a *separate table*.

11.10 Token Pattern Annotations

The Elsevier Fingerprint Engine has several activities generating `TokenPattern` annotations. Token Pattern annotations are RangeAnnotations that express that a range of tokens denotes a pattern of a certain type. Token Patterns are recognized by Regular expression annotators but also by the POS-Pattern annotation activity.

The conceptual meaning of Token Patterns is expressed by the `Name` property assigned to them, typically designating the class of items that the pattern belongs to, viz. 'NP' for noun phrases, 'copyright' for copyright notices in the text, etc.

The TokenPattern annotations also have an integer 'Score' property that designates the confidence or level of evidence available for this annotation; a lower value expresses that the annotator is not so certain that the given label [in Name] is the only correct label. There is no absolute maximum or threshold associated with the value, so the interpretation of the score is only possible relative to the other annotations available for the same stretch of text.

For instance, if you use a POS-Pattern finding activity for finding noun phrases, you could select the name 'NP', as in the example below, again from the input text `This is a sample text`.

```
<Annotation i:type="TokenPattern">
    <End>9</End>
    <Offset>6</Offset>
    <Name>NP</Name>
</Annotation>
```

CHAPTER

TWELVE

PART OF SPEECH TAGGING

Part of Speech tagging is an increasingly important activity in text analysis. Below is a table showing the possible tags the POS tagger can produce. The third column shows an example sentence, where the tag at hand is used. The corresponding word is *highlighted*

Tag	**Description**	**Example**
‘	quote	
*	negation	no, nor, not
ab	pre/post-qualifier	quite, rather, enough
at	article	a, the, no
be	be-verb	is, was
cc	coordination	and, but, or
cd	cardinal	one, two, three
cs	relativizer	if, although
do	do-verb	did, does
dt	determiner	this, that
ex	existential (‘there’)	there
hv	have-verb	has, had
in	preposition	of, in, by
inn	numerical preposition (:)	
jj	adjective	yellow
jjr	comparative adjective	bigger
LT	punctuation mark	
nn	noun	disease
nn$	possessive noun	dog’s
nns	plural noun	diseases
nns$	plural possessive noun	sisters’
np	proper noun	IBM
nps	plural proper noun	Carolinas
nr	directional noun (north)	North
od	ordinal	first, 2nd
pn	pronoun	you, they, she
ql	qualifier	very, quite, too
rb	adverb	quickly, never
rp	prepositional particle	through, at
to	to (infinitive to)	to
uh	fixed expression (yes)	ah, oops
vb	verb	eat
wdt	wh-determiner	which, that
wp	wh-pronoun	what, who
wql	wh-qualifier	how
wrb	wh-adverb	how, where

CHAPTER

THIRTEEN

WORKFLOW CONTROL BY LANGUAGE AND METADATA

The processing logic in the Fingerprint Engine is normally completely determined by selection of the workflow. However, there are two configurational possibilities to steer the workflow in a specific direction, viz.

- *control by language as determined from the input text*
- *control by metadata as submitted in a metadata section in the input*

13.1 Language Dependent Processing

The Elsevier Fingerprint Engine supports, on demand, multiple languages. To meet the need of users that do not know the language of input beforehand, the Elsevier Fingerprint Engine features a language recognizer. The results of the language recognizer is stored in a single annotation: a `LanguageAnnotation` that can be used to control the workflow.

13.1.1 Detecting the language of the input

The language of the input text can be detected by the *DetectLanguage* activity. This activity, if it is to control the workflow, should be the first activity in the workflow. The logic and complexity of this activity is embodied in a *language detector static object* as described below.

13.1.2 Controlling the Workflow by language

After language recognition, we must define different code paths for different languages. This is done by embedding partial workflows [sequences of activities] in a `LanguageDependentTextAnalysisActivity` in the following way:

```
<a:DetectLanguage         x:Name="DL"         Key="LR" />
<LanguageDependentTextAnalysisActivity Languages="default,en">
  <a:Tokenize               x:Name="TokenizeEN"   Tokenizer="en"  />
  <a:Normalize              x:Name="NormEN"     Normalizer="en" Score="0.9" />
  <a:DetectAbbreviations    x:Name="AbbrEN"       Normalizer="en"
                                                Language="en" />
  ...
</LanguageDependentTextAnalysisActivity>
<LanguageDependentTextAnalysisActivity Languages="de">
  <a:Tokenize               x:Name="TokenizeDE"   Tokenizer="de"  />
  ...
</LanguageDependentTextAnalysisActivity>
```

The dots [...] indicate that any number of activities may be embedded in the `LanguageDependentTextAnalysisActivity`. The `LanguageDependentTextAnalysisActivity` has one required attribute, viz. `Languages`. This attribute must contain a comma-separated list of *ISO-939-3 2-letter language codes* (see below) that determine the languages that this workflow applies to. If the `LanguageAnnotation` produced by the language recognizer is contained in this list, then the embedded activities will be executed.

13.1.3 Language Recognition Static Object

The Language Recognition static object contains a block of statistical data that enable the language recognition activity to determine the language of an input text. The Language Recognition object is available from the namespace `TACO.StaticObjects` and defined in the assembly `TACO.LanguageRecognition.dll` which is available upon request.

The currently most effective and recommended object that answers the `ILanguageDetector` interface is named `LanguageFingerprintsRecognizer`. This object can create a 'fingerprint' from a piece of text and discern what language it is by comparing that fingerprint against fingerprints generated from multi-lingual data in a trainingset.

The language recognizer takes two optional configuration parameters:

- SampleSize : an integer value limiting the size of the text sample that the LanguageRecognizer takes to recognize the language
- LimitTo : a comma-separated list of language codes that the language recognizer can detect.

```
<WorkflowRuntime.StaticObjects
        xmlns:l="clr-namespace:TACO.StaticObjects;assembly=TACO.LanguageRecognition">
    ...
    <!-- InterfaceType="ILanguageDetector" -->
    <NamedObjects>
        <l:LanguageFingerprintsRecognizer xaml:Key="CLR" LimitTo="es,en,pt,fr" />
    </NamedObjects>
    ...
</WorkflowRuntime.StaticObjects>
```

Below is a table showing the possible languages that the language recognizer can recognize.

Language	Name
da	Danish
de	German
el	Modern Greek (1453-)
en	English
es	Spanish
fi	Finnish
fr	French
it	Italian
nl	Dutch
pt	Portuguese
sv	Swedish
ja	Japanese
zh	Chinese

However, the Elsevier Fingerprint Engine software does not have full support for all of these languages. For most operations, support is available only for English, Dutch and German, and to varying degrees, French, Spanish and Portuguese. (On request, other language modules can be made available, however).

So, normally the user will want to limit the search of the Language Recognizer to the languages that are actually supported by modules. This can be done by e.g. assigning a value "de,en" to the `LimitTo` attribute. If the search is not limited, then a language outside of the scope of the available functionality may be detected. If language dependent workflows have not been defined for that language, the result may be that there is no output. (Of course this may be intentional and better than producing bad output.)

If no language can be recognized, then the value 'default' will be returned.

13.2 Metadata Filter Processing

13.2.1 Controlling the Workflow by metadata

The workflow can be controlled by metadata by referencing a defined `MetadataFilter`. The metadata filter can say 'yes' or 'no' depending on the metadata present. The workflow is controlled by the metadata filter by embedding optional processing steps in a wrapper step `MetadataDependentTextAnalysisActivity`.

Configuration is as follows:

```
<a:MetadataDependentTextAnalysisActivity MetadataFilter="msh" >
    <!-- annotate adverbial constituents -->
    <a:AnnotateTokenRanges  x:Name="AdverbialConstituents"
                            Annotator="AdverbialConstituents" NameProperty="name" />
    <a:HidePatterns         x:Name="HidePatterns_11"
                            Patterns="comma" IsBoundary="true" />
</a:MetadataDependentTextAnalysisActivity>
<a:MetadataDependentTextAnalysisActivity MetadataFilter="cpx" >
    <a:FindTerms            x:Name="Cpx"        Thesaurus="MetaCompendex"
                                                FilterMethod="Restrictive" />
</a:MetadataDependentTextAnalysisActivity>
```

A configuration such as the above determines that if the *metadata filter* called 'msh' decides 'yes' - that the processing steps 'AnnotateTokenRanges' and 'HidePatterns' be executed. Etc.

13.3 Metadata Filter Definition

The XmlMetadataFilter static object contains a list of associations between Xml nodes [nodes with textual content] and required text contained in that node. If an Xml node with the described content is found in a metadata section [as determined by a preprocessor], the filter returns 'yes' - otherwise 'no.'.

```
<WorkflowRuntime.StaticObjects  >
    <NamedObjects
        xmlns:se="clr-namespace:TACO.Core.Sections;assembly=TACO.Core">

        <!-- InterfaceType="IMetadataFilter" -->
        <se:XmlMetadataFilter xaml:Key="cpx">
            <se:XmlSectionContent Section="Workflow" Content="cpx" />
            <se:XmlSectionContent Section="workflow" Content="cpx" />
        </se:XmlMetadataFilter xaml:Key="cpx">

    </NamedObjects>
</WorkflowRuntime.StaticObjects>
```

The above configuration simply encodes that if a node `<workflow>stw</workflow>` is present, the filter says 'yes'.

CHAPTER

FOURTEEN

TEXT PATTERN ANNOTATORS

From one point of view, we can distinguish, in the Elsevier Fingerprint Engine, two methods of annotation: purely textual annotations and annotations based on other annotations. In the same vein there are two types of pattern matchers: *annotation pattern matchers* and text pattern matchers. This document outlines the capabilities and configuration of the various text-based pattern matchers.

The Elsevier Fingerprint Engine provides several means of annotating textual patterns:

- the RegexTokenPatterns,
- the NamedRegexPatterns,
- the RegexTokenPatternMatcher,
- the RegexNamedTokenPatternMatcher,
- the FixedPatternAnnotator,
- the FixedExpressionAnnotator

Each of them must be referenced by an *AnnotatePatterns* activity in order to be put to use.

14.1 RegexTokenPatterns

In the static objects we can define a pattern matcher that scans for the occurrence of regular expressions in the input text. The RegexTokenPatterns static object does exactly that. The `RegexTokenPatterns` object can be used by the *Annotate Pattern* activity. In its definition, it needs a regular expression pattern `Pattern` attribute and a `Name` attribute (assigned to inserted TokenPatterns in output). Defining patterns can be a very difficult process (particularly, debugging them). For that reason, we introduced a syntax to enable the user to split a complex pattern into parts, which is defined *below*.

```
<NamedObjects>
    <m:RegexTokenPatterns   xaml:Key="EmailAddresses"
                            Pattern="\b\w+@(\w+\.)+\w+\b"
                            Name="EmailAddress" />
</NamedObjects>
```

14.2 NamedRegexPatterns

The simple *Regular expression text matcher* matches a single pattern only. The downside of that is that it is not very efficient. We may want to define a regular expression that scans for different patterns of data in one single go, and outputs TokenPatterns with names depending on the pattern matches. The solution to that is in *named groups*. The C# regular expression syntax allows for naming matching groups as follows:

```
(?<date>[0-9]{2}-[0-9]{2}-[12][0-9]{3})
```

The NamedRegexPatterns objects use the names in the regular expression pattern defined, and outputs tokenpatterns accordingly. The example regular expression below outputs, if it matches, either a ‘figure’ string (e.g. “Fig. 1”) or a citation (e.g. “M. McCauley (2000)” or “A.N. Chomsky (1965)”). Note that non-capturing groups are explicitly marked with `(?:` - if that were not done, the output would contain many useless token patterns for these groups.

Defining patterns can be a very difficult process (particularly, debugging them). For that reason, we introduced a syntax to enable the user to split a complex pattern into parts, which is defined *below*.

```
<NamedObjects>
    <m:NamedRegexPatterns   xaml:Key="EmailAddresses" >
        <NamedRegexPatterns.Pattern>
            <![CDATA[(?<Figure>Fig\. [0-9]+)|(?<Citation>(?:[A-Z]\.)+
            (?:(?:Mc)|(?:O'))?[A-Z][a-z]+ \([1-2][0-9]{3}\))]]>
        </NamedRegexPatterns.Pattern>
    </m:NamedRegexPatterns>
</NamedObjects>
```

14.2.1 Defining Complex Regular Expression Patterns

Defining regular expression patterns can be a very difficult process (particularly, debugging them). For that reason, we introduced a syntax to enable the user to split a complex pattern into parts. We allow for the definition of `Snippets` that each define a regular expression pattern matching a meaningful unit. These snippets are defined as a member of the Regular-Expression based static object, either `NamedRegexPatterns` or `RegexTokenPatterns`.

The pattern matcher member `Snippets` is a dictionary of regular expression portions. The type of this dictionary is from string - to string Each entry in this dictionary consists of a *Key-Value* pair, defined as `<c:string xaml:Key="...">Value</c:string>`. Each dictionary entry defines a regular expression portion by name. The Xaml-syntax for defining a Snippets-member like this is as follows:

```
<m:NamedRegexPatterns Pattern="(?&lt;Name&gt;`ForwardFullName`|`ReverseFullName`)"
                 xaml:Key="IgnorePatterns">
    <m:NamedRegexPatterns.Snippets>
        <m:Snippets>
            <c:String xaml:Key="4digits">[0-9]{4}</c:String>
        </m:Snippets>
    </m:NamedRegexPatterns.Snippets>
</m:NamedRegexPatterns>
```

Snippets are referenced in the `Pattern` variable by their name surrounded with back-ticks (reverse quotes). Snippets can be defined in terms of one another, but recursive definitions are illegal. Definitions may reference one another to arbitrary depth but an exception will be thrown if a recursion is detected. In order to facilitate debugging and error detection, it is required that each snippet must be a legal regular expression on its own.

The MeSH-sample configuration delivered with the Elsevier Fingerprint Engine sample configurations contains an elaborate definition of (safe) citation patterns as found in scientific literature. As an instructive example a part of that definition is repeated here.

```
<m:NamedRegexPatterns Pattern="(?&lt;Name&gt;`ForwardFullName`|`ReverseFullName`)"
                 xaml:Key="IgnorePatterns">
    <m:NamedRegexPatterns.Snippets>
        <m:Snippets>
            <c:String xaml:Key="Surname">(?:(?:Ma?c)|(?:O\'))?\p{Lu}\p{Ll}+(?:-\p{Lu}\p{Ll}+)?</c
            <c:String xaml:Key="ReverseFullName">`Surname`,? `Initials`</c:String>
            <c:String xaml:Key="ForwardFullName">`Dotted` `Surname`</c:String>
            <c:String xaml:Key="Initials">(?:`Dotted`|[A-Z]{1,3})</c:String>
            <c:String xaml:Key="Dotted">(?:(?:[A-Z]\. ?){1,3})</c:String>
        </m:Snippets>
    </m:NamedRegexPatterns.Snippets>
</m:NamedRegexPatterns>
```

14.3 RegexTokenPatternMatcher

TODO

14.4 RegexNamedTokenPatternMatcher

TODO

14.5 FixedPatternAnnotator

The FixedPatternAnnotator lives in the namespace `xmlns:e="clr-namespace:TACO.Core.EntityAnnotator;assem` which must be mentioned in the XML pre-amble of the Runtime Configuration. The annotator offers the functionality to annotate a number of pre-defined patterns, as well as matching a pre-set, fixed list of simple text strings as defined in an external data source.

The following patterns can be recognized:

- Email: simple email addresses of the type *john@doe.com*
- Pagination: simple number-number patterns of the type *19-29*
- Roman: roman numbers (IV, vii, MCMLXXXVI)
- Words: words or string expressions defined in an external *Wordset* or file, defined by `FilePath` in the configuration

The type of pattern matcher must be defined in the `Pattern` field on the Xaml configuration. If `Words` is selected, additionally either a `Words` field must be defined [referring to *Wordset*] or `FilePath`. For all patterns a score-factor can be defined that assigns a score to each output *TokenPattern* annotation.

```
<NamedObjects>
    <!-- fixed pattern annotator to detect email addresses -->
    <e:FixedPatternAnnotator xaml:Key="Email" Pattern="Email" Name="email" Factor="10" />

    <!-- fixed pattern annotator to detect copyright sequences -->
    <e:FixedPatternAnnotator xaml:Key="CopyrightPatterns" Tokenizer="en"
                    Pattern="Words" FilePath="CopyrightPatterns.zip"
                    Name="copyright" Factor="10" >
        <e:FixedPatternAnnotator.Macros>
            <e:Macros>
                <e:Macro PlaceHolder="9999" Type="Year" />
            </e:Macros>
        </e:FixedPatternAnnotator.Macros>
    </e:FixedPatternAnnotator>
</NamedObjects>
```

14.6 FixedExpressionAnnotator

TODO

CHAPTER

FIFTEEN

MATCHING EXPRESSIONS

The Elsevier Fingerprint Engine features a number of activities that perform pattern matching. Patterns can be found in the text by the rather well-known system of regular expression matching. However, the Elsevier Fingerprint Engine offers the powerful capability of matching patterns on ranges [sequences] of annotations.

It is important to understand the notion that *patterns* are matched on input: annotations or text. Patterns are written down in *expressions*. These expressions are stored as *static objects*, for the reason that they will be compiled into executable patterns only once. The execution of patterns - i.e. the pattern matching, is done during execution of a workflow by an *activity*. Normally, each activity type can execute only one type of expression.

To manage the matching capabilities, the syntax of expressions has been left as simple as possible, and the various pattern matchers, as much as possible, adhere to general syntactic features.

This document outlines the global syntax of all expression types and gives details for specific types of expressions.

The table below offers the index of this document: the matching activity type - expression type pairs and the output produced by the activities.

Activity	PatternMatcher
AnnotateTokenRangePatterns	ITokenRangeMatcher
AnnotateTokenRanges	*TokenRangeRegularExpression*
AnnotatePOSPatterns	POSPatternMatcher
	POSPatternMatcher
AnnotateTerms	TermAnnotator
	TermAnnotator

15.1 General Expression Syntax

The matching expressions in the Elsevier Fingerprint Engine all share a common structure. In order to keep the expression readable, the user can define sub-patterns in the course of the definition of expressions. This not only allows for defining sub-expressions and thus keep an overview of the whole; it also enables the user to re-use sub-expressions at different locations in the main expression.

The general structure of an expression as it occurs in the *RuntimeConfiguration* of the Fingerprint Engine definition is as follows:

```
<e:EXPRESSIONTYPE Expression="TOPNODE" xaml:Key="REFERENCENAME" Matching="Maximized|Exhaustive" >
   <e:EXPRESSIONTYPE.Definitions>
     <c:String>
                 TOPNODE = A B C ;
                     A = ... ;
                     B = ... ;
                     C = D E ;
                     D = ... ;
                     E = ... ;
     </c:String>
   </e:EXPRESSIONTYPE.Definitions>
 </e:EXPRESSIONTYPE>
```

The following **ALL CAPITAL** variables are used:

- EXPRESSIONTYPE : type of expression (*TokenRangeRegularExpression* or others)
- TOPNODE : the expression that is to be matched
- REFERENCENAME : the name to use to reference the expression in the activity
- Matching : either Maximized or Exhaustive
  - Maximized: once the expression matches, restart at the first position *after* the match
  - Exhaustive: match at all possible positions

The definition of the top-node expression and the sub-expressions is contained in a 'Definitions' sub-node in the Xml. The syntax of the definitions is simple : each expression is defined as an assignment **A = B ;**. Lines must end in a semi-colon. If one uses sub-expressions, these subexpressions can be reference by an alphanumerical name. This alphanumerical name must stand apart from its context by whitespace. (After the entire expression has been put together, the whitespace is 'squeezed out' - no whitespace can be matched by this type of expression - at least not by whitespace). Definitions of sub-expressions cannot be recursive, i.e. an expression cannot be defined in terms of itself - a definition of this type would blow up the expression upon expansion and brings the expression outside of the finite state realm. So: **" A => B C; B => A C; "** is illegal and the Fingerprint Engine will throw an exceptions if it detects a pattern like that.

If one looks at a set of rules as a Backus Naur rules, this general outline gives you the combinary syntax of the intermediate, nonterminal nodes only; the syntax for the terminal nodes of the expression is specific for the type of expression.

15.1.1 Quantification

Each terminal and non-terminal node in expressions can be quantified by the same syntax as in usual regular expressions. In other words, one can write (**X** as the placeholder for a terminal or non-terminal):

- **X+** : match 1 or more times
- **X?** : match 0 or 1 time
- **X*** : match 0 or more times
- **X{n,m}** : match n to m times

15.1.2 Grouping

Each terminal and non-terminal node in expressions can be grouped by the same syntax as in usual regular expressions. In other words, one can write (**X** and **Y** as placeholders for terminals or non-terminals):

- **(XY)** : match X and Y
- **[XY]** : match X or Y
- **(?<name>XY)** : match X and Y in a *named* group

What the expression matcher does with named groups depends of the activity. The activity **AnnotateTokenRangePatterns** will produce sub-patterns in the overall **RangeAnnotation** output and create a **Class** entry for each subpattern, containing the name of the group and the contents of the group.

15.2 TokenRangeRegularExpression

A **TokenRangeRegularExpression** is an expression that can range over all annotations that answer to the interface **ITokenRange** - in other words, that span a range of tokens.

The combinatory syntax of these expression is as outlined *above*. The *terminals* of the TokenRangeRegularExpression are TokenRanges. Currently the following types of TokenRanges can be matched:

- W : IWord
- T : TermAnnotation
- P : POSTag

The capital letters above introduce the type of each TokenRange terminal.

15.2.1 General syntax of terminals

The general syntax for defining terminals is **X/.../** - an initial letter (either of W, T, P) and further definitions between slashes **//**. If there are no further definitions, *all* ranges of that type will match. Between the slashes, Annotation properties of **Integer**, **Flags** and **String** types can be checked. Each annotation type (IWord, TermAnnotation, POSTag) has its own specific properties. For IWord, these properties are (for example) **SourceID** and **Text**; for TermAnnotation the most relevant properties are **Flags**, **ConceptID** and **TermID**.

The syntax for checking properties is straightforward:

- X/PROPERTY=1/ : match an integer property with value 1 - all other X annotations are rejected
- X/PROPERTY!=1/ : match an integer property with value *not* 1 - all X annotations with PROPERTY=1 are rejected
- X/PROPERTY='xyz'/ : match a string property with value 'xyz' or a Flags property containing a flag 'xyz'
- X/PROPERTY!='xyz'/ : match a string property with value not 'xyz' or a Flags property not containing a flag 'xyz'
- X/PROPERTY~'xyz'/ : match a string property matching regular expression 'xyz'
- X/PROPERTY!~'xyz'/ : match a string property not matching regular expression 'xyz'

Property checks can be combined by putting a semicolon ';' in between.

Examples: * W/SourceID=1/ : match a normalized form only (SourceID=1 means: from Normalizer 1) * W/SourceID=0/ : match a original word form only (SourceID=0 means: from the original text) * T/Thesaurus='MeSH';Flags~'^C'/ : match a term annotation originating from Thesaurus MeSH only, containing a flag matching the regular expression **^C** (meaning: starting in **C**)

15.2.2 W : IWord Expressions

Apart from the general syntax for property checks, each expression starting with **W** can have the following specific syntactic tweaks.

- Between single quotes, one can specify the text that the word must match: **W/'of'/** will match the word 'of' only. This is shorthand for **W/Text='of'/**.
- Between square brackets, one can specify a (comma-separated) list of texts that the word must match:
  - **W/['of','in','on']/** will match either 'of', 'in' or 'on'.

15.2.3 T : TermAnnotation Expressions

TermAnnotation terminal definitions have no special syntax for property checks (just the general syntax). However, upon matching a TermAnnotation we can extract special properties by specifying these properties between curly braces. These properties will be put in the output as classes to the sub-phrase extracted.

- **T{ConceptID,TermID}/Thesaurus='MeSH'/** will extract the **ConceptID** and **TermID** from the TermAnnotation.

15.2.4 P : POSTag Expressions

Apart from the general syntax for property checks, POSTag expressions have the following specific syntactic tweaks. A simple tag (as by the *list*) between the slashes will match the tag. Specifying a comma-separated tag within square brackets selects the more detail SemanticTags (not documented). Specifying a string between single quotes will match a text corresponding to the range that the POSTag annotation covers.

- **P/[vb,vbz]/** : match infinitive or third person singular present tense verb forms
- **P/vb/** : match a verb
- **P/’of’in/** : match a preposition **in** with the text ‘of’.

15.2.5 Example TokenRangeRegularExpression

```
<e:TokenRangeRegularExpression Expression="Relation" xaml:Key="Relation" Matching="Maximized" >
  <e:TokenRangeRegularExpression.Definitions>
    <c:String>
     Relation =  [ UnconstrainedRelation DisDisRelation ];
     UnconstrainedRelation = (?&lt;Relation&gt; (?&lt;Term1&gt; Term ) Connector (?&lt;Term2&gt;
     DisDisRelation  = (?&lt;Relation&gt; (?&lt;Term1&gt; DisTerm ) CAUSE (?&lt;Term2&gt; DisTerm
     Term = T{ConceptID,TermID}/Thesaurus=’MeSH’;Flags!=’fraction’/ ;
     DisTerm = T{ConceptID,TermID}/Thesaurus=’MeSH’;Flags~’^C’/ ;
     Connector = [ Induced Compound Containing ];

     Induced    = (?&lt;induced,relation=induced&gt;W/’-’/{0,2}W/’induced’/) ;
     Containing = (?&lt;containing,relation=containing&gt;W/’-’/{0,2}W/’containing’/) ;
     CAUSE      = (?&lt;from,relation=cause&gt;W/’from’/) [ P/at/ P/dt/ ]? P/qb/* P/jj/*;
     Compound   = (?&lt;compound,relation=compound&gt; ) ;
    </c:String>
  </e:TokenRangeRegularExpression.Definitions>
</e:TokenRangeRegularExpression>
```

15.3 TermAnnotator static object

For the purpose of rule-based disambiguation, a specific disambiguator static object has been derived from the TokenRangeRegularExpression static object. This `TermAnnotator` static object can be optimized for disambiguating terms; since the targets of the pattern matching are strictly terms, the underlying text and annotations can be scanned first for possibly applicable terms, before the actual pattern is executed.

This annotator is referenced by the *AnnotateTerms* activity.

```
<!-- context rules handling disambiguation -->
<e:TermAnnotator Expression="BadCompound" xaml:Key="Disambiguate" Matching="Maximized"
                        CheckDefinitions="true" SpeedUp="BranchIndex"
                        ExternalPath="Disambiguation.xaml" />
```

15.4 POSTagPatternMatcher static object

The Elsevier Fingerprint Engine features a special pattern matcher that can extract patterns of POS (Part-of-Speech) tags in the text.

The POSTagPatternMatcher needs `POSTags` as input so an activity producing those tags must be executed first.

The Pattern to be matched is defined in the `Pattern` attribute. This pattern is essentially a regular expression pattern; however, contrary to the commonly used regular expressions the input is not the character array (string) input, but the `POSTag` annotation array; each unit of the input is not a character but a POSTag. Units are defined in the pattern by the syntax `<...>`. Between the angled brackets, any property of the tokenrange type at hand can be referenced by the `Property="..."` syntax, e.g. `<Tag="jj">` means: match a POSTag with the tag value `jj` (which means: adjective). The text of the token range can be matched by the `<"text">` syntax, which means that only token ranges with that text must be matched. Aside from this notation of units, all other regular expression syntax constructs are allowed, including

- `*` for 0 or more repetitions
- `+` for 1 or more repetitions
- `{n,m}` for n to m repetitions
- `()` for grouping
- `[]` for any-of groups
- `?` for 0 or 1 occurrence

Since < angled brackets are disallowed in XML-attributes, the &lt; and &gt; notation must be used. Due to the combination of the syntax of POSTagPatternMatcher patterns and the XML constraints, the notation of POSTagPatternMatcher patterns can become a real notational nightmare. Defining entities in the XML document DTD may take away some complications, but to avoid that type of complexity we devised a schema where `.nn.` can be used as a shorthand for `<Tag="nn">`. Nevertheless, writing patterns is a task that needs expert input in most cases. To use this type of pattern, use the `POSPatternMatcher` static object type, which is a derivative of the `POSTagPatternMatcher`.

15.4.1 POSPatternMatcher

Using the `POSPatternMatcher` enables us to write a noun-phrase pattern as follows.

```
<NamedObjects>
    <m:POSPatternMatcher
            xaml:Key="NPs"
            Pattern = '[.jj..nn..nns..np.]*(.jj..in.)?[.jj..nn..nns..np.]*[.nn..nns..np.]'
            Name="NP" />
</NamedObjects>
```

The POSPatternMatcher can be referenced by *Part-of-Speech pattern annotating* activities.

15.5 TokenPatternMatcher static object

Token pattern matchers have been defined for internal purposes only and are not documented for external use.

CHAPTER

SIXTEEN

QUERYING THE ELSEVIER FINGERPRINT ENGINE

16.1 The Fingerprint Engine query handle

If a Fingerprint Engine instance has successfully been installed on a host machine (say, localhost), the Fingerprint Engine exposes several handles that can be called for information.

The Fingerprint Engine provides text analysis services based on thesauri or controlled vocabularies. For external products, we may need access to two more types of information about Thesaurus concepts. Specifically, two queries are required:

- Concepts: a query to retrieve Concepts IDs and Names from the thesaurus
- Terms: a query to retrieve terms by prefix (-es)
- ConceptList: a query to retrieve all concepts *and* their semantic groups
- IgnoreConcepts: a query to retrieve all ignore concepts
- SuspiciousConcepts: a query to retrieve all suspicious concepts

16.1.1 REST Interface

The data is accessed by a REST interface, implemented using WCF like the other Fingerprint Engine interfaces. A specialized handle called `Query.svc` deals with queries of different types. Currently `Terms`, `Concepts`, `ConceptList`, `Hierarchy`, `IgnoreConcepts` and `SuspiciousConcepts` queries have been implemented.

Query for Search by Terms	<baseurl>/Query.svc/Terms/{thesaurus}/{maxTerms}
Query for Concept Information	<baseurl>/Query.svc/Concepts/{thesaurus}
Query for Listing Concepts	<baseurl>/Query.svc/ConceptList/{thesaurus}
Query for IgnoreConcepts	<baseurl>/Query.svc/IgnoreConcepts/{thesaurus}
Query for Hierarchy	<baseurl>/Query.svc/Hierarchy/{thesaurus}
Query for Suspc. Concepts	<baseurl>/Query.svc/SuspiciousConcepts/{thesaurus}

For all services, input information is sent by the POST method. A piece of Sample client code is provided.

16.2 Input

16.2.1 Input for Terms query

This is a POST query - you must post the search term after the URL. The terms query will parse the input text as words, separated by whitespace. A minimum of three non-whitespace characters are required; if this requirement is not met, an error is returned. The maxTerms parameter [part of the URL] is parsed as an integer; if this is not a valid integer, an error is returned. Valid input strings are: “af mos”, “a b c”; invalid input strings are: “”, “a b” or “ab”. Control characters and other garbage are treated as whitespace and not reported.

Example URL: https://fingerprintengine.ptgels.com/TACO7500/Query.svc/Terms/MeSH/20

POSTing 'blood', you receive the return value:

```
<Result>
    <Concept ID="17" Name="ABO Blood-Group System"><Term>H Blood Group</Term></Concept>
    <Concept ID="2669" Name="Blood"><Term>Blood</Term></Concept>
    <Concept ID="2673" Name="Blood Banks"><Term>Blood Banks</Term></Concept>
    <Concept ID="2677" Name="Blood Cells"><Term>Blood Cells</Term></Concept>
    ...
    <Query>blood</Query>
</Result>
```

16.2.2 Input for Concepts query

The concepts query treats the input text as a comma-separated list of integers. If the parsing of the input text fails somewhere, this is reported.

Example URL: https://fingerprintengine.ptgels.com/TACO7500/Query.svc/Concepts/MeSH

POSTing '17,2669,2673', you receive the return value:

```
<Result>
    <Concept ID="17" Name="ABO Blood-Group System"/>
    <Concept ID="2669" Name="Blood"/>
    <Concept ID="2673" Name="Blood Banks"/>
    <Query>17,2669,2673</Query>
</Result>
```

16.3 Output for queries

16.3.1 Return Xml format

All information returned is in sane Xml format. The structure of the return value for all queries is

```
<Result>
    <Error> [Error message] </Error>*
    <ReturnedItem />+
    <Query> [ the original query text [POSTed] </Query>
</Result>
```

This means that the top node is always `Result`. There may be `<Error>` nodes containing a message; if so, the results will not necessarily be valid. There will always be an embedded node `<Query/>` containing the originally POSTed query. The tag and contents of the ReturnedItem nodes varies according to the query.

16.3.2 Terms query result

The terms query returns the concepts associated with a search string. All words in the search string must match a word in a term associated with a concept in the thesaurus. Casing is ignored. So, "fi fl" as well as "FI FL" will match "Fire Fly", etc. The words matched will be the *normalized* words, so "fi fli" will not match "Fire Flies" - because "Fire Flies" gets normalized to "fire fly". Multiple words in the search term must match multiple words in the term. So, "fi fi" can match "fiscal fidelity" but not "fiscal". Multiple tokens in the input must match multiple words in the matched terms!

The result contains of a number (maximally <maxResults>) of nodes of the following format, replacing the <ReturnedItem /> placeholder as outlined above:

```
<Concept Name="XYZ" ID="123">
    <Term>[term text]</Term>+
</Concept>
```

All concepts associated with terms found are returned. If multiple terms of a single concept match the search query, all of these are returned.

16.3.3 Concepts query result

The concepts query result contains concept nodes of the following format, containing names for the ids present in the query.

```
<Concept ID="<integer id>" Name="Concept display name" />
```

Missing concept IDs [queried but not present in the thesaurus] will be reported in an error message.

16.3.4 Client code

Elsevier provides client code that can

- Send queries
- Translate returned Xml into C# objects

We shall define a class `Concept` that may contain terms, but at least contains the members ConceptID and DisplayName. The returned objects will look like: * For Terms: List<Concept> * For Concepts: List<Concept> The client code is made available as a library (`TACO.Client.dll`).

CHAPTER

SEVENTEEN

MANAGING THE ELSEVIER FINGERPRINT ENGINE

17.1 The Fingerprint Engine management handle

If a Fingerprint Engine instance has successfully been installed on a host machine (say, localhost), the Fingerprint Engine exposes a handle that can be called for managing an installation.

17.1.1 REST Interface

The data is accessed by a REST interface, implemented using WCF like the other Fingerprint Engine interfaces. A specialized handle called `Management.svc` deals with commands of different types. Currenlty, the only command is `Reload/Thesaurus`. To call it, do a GET request on:

Reload a thesaurus	<baseurl>/Management.svc/Reload/Thesaurus/{thesaurus}

[This is implemented as a GET method to make it possible to trigger the command from a web browser's command line]

17.2 Effect of the Reload/Thesaurus command

The Reload/Thesaurus command will *not* reload the XAML from disk, but it will reread any data that it references.

So, one can use the command to load changed thesaurus data from a database, but not to load data using a different connection string.

Also, to make changes effective in a workflow, one may have to call the same Reload/Thesaurus multiple times. For example, when one reloads a thesaurus that is part of a MetaThesaurus, one must reload the metathesaurus for it to pick up the changed data.

17.3 Output for the Reload/Thesaurus command

17.3.1 Return Xml format

All information returned is in sane Xml format. A succesful call returns:

```
<Result>OK</Result>
```

A failing call returns:

```
<Result>
    <Error> [Error type]
        <Message> [Error Message] </Message>+
            </Error>
</Result>
```

This means that the top node is always `Result`. There may be multiple `<Error>` nodes; if so, messages later in the file give lower-level information about the reason for the error. Two examples:

```
<Result>
        <Error>
                ApplicationException
                <Message>Could not find a static object named 'blabla' of type IThesaurus</Messag
                    <Message>Known static objects of type IThesaurus:</Message>
                    <Message>'CopyrightPatterns' (a FixedPatternAnnotator)</Message>
                    <Message>'Email' (a FixedPatternAnnotator)</Message>
                    <Message>'Idiom' (a SQLCKEThesaurus)</Message>
                    <Message>'MeSH' (a SQLCKEThesaurus)</Message>
                    <Message>'MeSHExtensions' (a GenericThesaurus)</Message>
                    <Message>'MetaMeSH' (a MetaThesaurus)</Message>
                    <Message/>
            </Error>
    </Result>

<Result>
        <Error>
                ApplicationException
                    <Message>Could not initialize cloned object named 'MeSHExtensions'</Message>
        </Error>
            <Error>
                TargetInvocationException
                    <Message>Exception has been thrown by the target of an invocation.</Message>
            </Error>
            <Error>
                TACOException
                    <Message>TACO #SystemError : If concept IDs are specified, please note that
                    they must be integer: Concept ID 231d56 for term blabla is invalid.</Message>
            </Error>
    </Result>
```

CHAPTER

EIGHTEEN

ADAPTERS

Adapters are a technical construct to constrain the application of *activities*.

For instance, we might want to limit term finding to noun phrase generation to text areas that are not citations or copyright notices.

In this chapter all available adapters are described in more detail. For each adapter we list:

- the activities they can meaningfully applied to
- the functionality of the adapter
- an instantiation example of the adapter

Select Terms Select terms to be processed, by thesaurus name associated with the term.

Select Sections Select sections to be processed, by section name assigned to the section.

Block Out Patterns Rule out specific patterns from consideration by the activity.

Block Out Terms Rule out specific terms from consideration by the activity.

18.1 Select Terms

`SelectTerms`

18.2 Select Sections

`SelectSections`

18.3 Block Out Terms

In case one performs term finding with two thesauri covering different domains, or two thesauri with on thesaurus (A) taking precedence over the other (B), we can *block out* terms from thesaurus A from the input to the term finding activity referring B. The *FindTerms* activity implicitly requires the input of `Sentences`. If this attribute is not set, all sentences that are present in the data structure are search for terms. If one assigns the `BlockOutTerms` adapter to this attribute, then only sentences or portions of sentences that are not known as Terms form the input to the `Sentences` handle.

This adapter is specified as follows:

```
<a:FindTerms x:Name="SecondTimeTermFinding"
             Thesaurus="secondThesaurus"
             MaxGapSize="1"
             Sentences="{s:BlockOutTerms Name=firstThesaurus}"
             FilterMethod="Restrictive|PreferBetter" />
```

If the `Name` variable on the `BlockOutTerms` adapter is not mentioned, then *all* previously found terms will be 'blocked out'. To 'block out' a term not only means that the stretch of text that this term covers is not searched for new terms; it also implies that the sentence is actually split at that point. For instance, if the input sentence reads 'This is my nice and long input sentence' and 'nice and long' is a concept in the first thesaurus; then applying the `BlockOutTerms` adapter yields *two* sentences, viz. 'This is my' and 'input sentence'. The blocked-out term is never even considered by the term finder reading the adapted input.

18.4 Block Out Patterns

In case one performs phrase generation, extracting phrases or features based on (for example) POS-patterns, one may wish to *block out* specific patterns recognized by pattern recognition. This is done by one of the pattern match activities, e.g. AnnotateClassPatterns. Similar to the *FindTerms* activity, this activity implicitly requires the input of `Sentences`. If this attribute is not set, all sentences that are present in the data structure are search for POS patterns. If one assigns the `BlockOutPatterns` adapter to this attribute, then only sentences or portions of sentences that are not known as Patterns form the input to the `Sentences` handle.

This adapter is specified as follows:

```
<a:AnnotateClassPatterns    x:Name="Phrases"
        Sentences="{s:BlockOutPatterns Patterns=Citation;Figure;copyright;email}"
        Annotator="Phrases"
        Normalizer="enp" />
```

If the `Patterns` variable on the `BlockOutTerms` adapter is not mentioned, then *no* previously found pattern will be 'blocked out'. To 'block out' a pattern follows the same logic of the *Block out Terms* adapter.

ELSEVIER

ELSEVIER