D11.1.1.b: Concept and Design of the Integration Framework

Workpackage: WP11
Deliverable ID: D11.1.1.b
UC-Name: ORDO
Document-Version: V1.0
Last Changes: 25.09.08
Authors (Organisations): Thomas Schütz (empolis GmbH)
Status: Final
Dissemination: Public
Reviewer: Ralph Traphöner, Igor Novakovic (empolis), Oliver Niese (empolis), Mario Lenz (empolis), Björn Decker (empolis)

ORDO is funded by the German Federal Ministry of Economics and Technology (grant number 01MQ070059 as part of the research program Theseus. Responsibility for the content of this publication lies solely with the author.
Summary

The project SMILA (Semantic Information Logistics Architecture) was founded to realize an Open Source framework for building search solutions to access unstructured information in the enterprise. This framework will provide an integration platform for a huge number of services from different vendors.

This Deliverable discloses the basic software-technical concept of SMILA. Chapter 2 gives an overview of the main objective, the basic requirements and the architecture of SMILA. In chapter 3 the most important concepts are described in detail.
### History

<table>
<thead>
<tr>
<th>Datum</th>
<th>Ver.</th>
<th>Author</th>
<th>History change</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.02.2008</td>
<td>0.1</td>
<td>Björn Decker</td>
<td>Document created</td>
</tr>
<tr>
<td>01.07.2008</td>
<td>0.17</td>
<td>Thomas Schütz</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; readable version</td>
</tr>
<tr>
<td>02.07.2008</td>
<td>0.18</td>
<td>Thomas Schütz</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; reviewed version</td>
</tr>
<tr>
<td>02.07.2008</td>
<td>0.19</td>
<td>Thomas Schütz</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; readable version</td>
</tr>
<tr>
<td>02.07.2008</td>
<td>0.20</td>
<td>Thomas Schütz</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; reviewed version</td>
</tr>
<tr>
<td>25.07.2008</td>
<td>0.21</td>
<td>Thomas Schütz</td>
<td>3&lt;sup&gt;rd&lt;/sup&gt; reviewed version</td>
</tr>
<tr>
<td>18.09.2008</td>
<td>0.22</td>
<td>Thomas Schütz</td>
<td>4&lt;sup&gt;th&lt;/sup&gt; reviewed version</td>
</tr>
</tbody>
</table>
Table of contents

1. Introduction ........................................................................................................................................ 6
   1.1 Project ORDO and SMILA ................................................................................................................. 6
   1.2 Participating parties ......................................................................................................................... 7
2. Overview of SIMLA ............................................................................................................................. 8
   2.1 Main objectives on SIMLA ............................................................................................................... 8
   2.2 Requirements concerning SIMLA .................................................................................................... 8
   2.3 Overview about the architecture and components ........................................................................... 9
   2.4 Related Technology .......................................................................................................................... 11
3. Architectural Concepts of SIMLA ....................................................................................................... 13
   3.1 Concepts for processing of records ................................................................................................. 13
      3.1.1 ID Concept ................................................................................................................................. 13
      3.1.2 Structured ID object ................................................................................................................. 15
      3.1.3 Record Data Model and XML representation ........................................................................... 18
         3.1.3.1 Logical Data Model in XML ............................................................................................... 20
      3.1.4 Blackboard Service concept ..................................................................................................... 23
      3.1.5 Router and Listener Concept .................................................................................................... 29
         3.1.5.1 Record Filter Concept ......................................................................................................... 30
      3.1.6 BPEL Pipelining Concept .......................................................................................................... 31
   3.2 Connectivity Module ......................................................................................................................... 34
   3.3 Information Reference Model (IRM) ............................................................................................... 37
   3.4 XML Storage Service Concept ....................................................................................................... 40
   3.5 Concepts of Infrastructure ................................................................................................................. 42
      3.5.1 Configuration Management ....................................................................................................... 42
      3.5.2 Monitoring ................................................................................................................................. 43
      3.5.3 Performance Measurement Framework ...................................................................................... 45
4. Conclusion ........................................................................................................................................... 47
5. Literature ............................................................................................................................................... 48
6. Glossary .................................................................................................................................................. 49
Figure Index

Figure 1: Architecture Overview ................................................................. 10
Figure 2: Workflow Processing using Blackboard Service ................................ 24
Figure 3: Sequence diagram Workflow Integration and Blackboard Service .......... 25
Figure 4: Blackboard Service with separated service ........................................ 28
Figure 5: Router Configuration Description .................................................... 30
Figure 6: Connectivity Module ...................................................................... 34
Figure 9: Monitoring architecture .................................................................. 44
Figure 10: Monitoring architecture in detail ..................................................... 45
1. Introduction

1.1 Project ORDO and SMILA

According to the requirements of ORDO an integration framework has to be designed and implemented. On this integration framework many components from different vendors must be integrated at low cost. In another study in this Work Package (WP11.1.1.a s. [Deliverable D11.1.1.a]) different integration frameworks are compared and evaluated with respect to their applicability in ORDO. None of these frameworks was considered to be satisfactory when compared to the SMILA approach. Either they were too expensive or they did not deliver an appropriate application for building search solutions to access unstructured information in the enterprise.

The study showed clearly the need for a reliable, standardized industrial strength framework to build search solutions to access unstructured information in enterprises. To address this need, the Open Source project SMILA (Semantic Information Logistics Architecture) was founded1.

To reach a wide acceptance in a developer community and to produce an attractive offer, competent partners were acquired (see chapter 1.2). In addition, SMILA has passed the “Creation Review” milestone of the Eclipse Development Process2 and is now an official project in incubation phase of the eclipse foundation3.

This Deliverable discloses the fundamental concept of SMILA. Chapter 2 gives an overview of the main objective, the basic requirements and the architecture of SMILA. In chapter 3 the most important concepts will be described in detail.

1 http://www.eclipse.org/smila/.

2 http://www.eclipse.org/projects/dev_process/development_process.php#6_3_1_Creation_Review

1.2 Participating parties

Initially participating parties of SMILA witch provide an initial code contribution:

- empolis GmbH⁴,
- brox IT-Solutions GmbH⁵,
- DFKI - German Institute for Artificial Intelligence⁶

As part of the Theseus, the SAP AG has decided to take up SMILA in the TEXO use case.

⁴ http://www.empolis.com/
⁵ http://www.brox.de/
⁶ http://www.dfk.de/
2. **Overview of SIMLA**

This chapter gives an overview of the main objective, the basic requirements and the architecture of SIMLA.

2.1 **Main objectives on SIMLA**

The main objective of SIMLA is to define and implement an extensible framework based on SOA principles and standards (e.g. BPEL, SCA), which is dedicated to building search solutions to access unstructured information in the enterprise. For this purpose, SIMLA provides essential infrastructure components and services as well as “ready-to-use” add-on components (e.g., connectors to data sources). Using the framework as their basis, developers are enabled to focus on creating higher value, semantic driven applications. Infrastructure features that are relevant to run semantic search applications solutions in enterprises (e.g. monitoring) will be provided by the platform.

The long term objective of SIMLA is to establish an industry standard by attracting as many parties as possible to use the framework and/or participate in the surrounding eco-system.

2.2 **Requirements concerning SIMLA**

The following requirements on SIMLA have to be considered (s. [Novakovic 2008]):

- Componentization: A major focus of SIMLA will be on componentization of the overall system architecture, thus ensuring that other open source tools, products by different vendors or even project-specific extensions can easily be plugged into the system.

- Exemplary implementation of vertical use cases like search, classification, text extraction, text annotation and other semantic analysis functions

- Data Source Management (integration and access): The objective is to make available a set of connectors (crawlers) for the most relevant data sources (e.g. file system, database systems and Web).

- Management, Operation & Monitoring: SIMLA will provide interfaces to allow for system management, monitoring and operation of its components
• Authentication and Authorization support: The end-users can interact with the system only according to their actual access rights. This is not only true for accessing and storing the information but also for the process execution within the framework.

• Status and performance reporting: Analytics and business intelligence reporting are essential parts of any Information Access Management (IAM) system. The information provided by the system not only allows to optimize its usage but also to identify missing information via knowledge gap analysis and similar approaches.

• Deployment on inexpensive hardware: Hardware nodes used for deployment of SMILA should not exceed the capabilities of a contemporary normal PC. More precise: The use of 1Gbit/s network adapter should be completely sufficient. A SMILA-process must have a small memory footprint.

• Scalability: The framework must be capable of handling huge amounts of data. The goal is to be able to deal with one billion documents and more.

• Reliability: Careful deployment planning and configuration of SMILA by e.g. avoiding single points of failure must ensure that the operation of SMILA will not be interrupted if some of its core components are suddenly not available.

• Robustness: Some bad component, misbehaving by taking 100% of CPU time or utilizing large amounts of memory, should not have an impact on the overall framework stability.

• Data consistency: Persisted application data must be consistent at any time. No matter what happens: power outage; the loss of complete network connectivity; total hardware failure; crash of all instances of a service - the data stored in the framework must not be corrupted.

• “Ease of use”: In order to reduce the amount of effort for utilizing SMILA some actions in community and partner readiness direction must be taken. The documentation of best practices and use case recommendations should be a part of the SMILA distribution

### 2.3 Overview about the architecture and components

The following figure gives an overview of the architecture and the most important components in SMILA [Novakovic 2008].
The purple boxes labelled “OSGi” represent OSGi runtime environments in which parts of SMILA can reside. However, this picture shows only one simple exemplary deployment scenario. Lots of other deployment configurations are possible (e.g., SMILA running in one OSGi runtime).

The queue buffers documents and other information. In this picture, it is used to buffer information provided by crawling data sources (upper left OSGI Box), which are processed afterwards (lower left OSGI Box).

The light blue boxes labelled BPEL contain exemplary pipelines for indexing and searching information.

**Architecture Overview**

![Architecture Overview Diagram]

**Indexing:**
- **Crawler** crawls the Data source and hands out the gathered data to the Connectivity module.
- **Connectivity** pushes the information into the Queue server.
- **BPEL** engine listens to the queue and consumes the messages.
- **BPEL services Text Export and Text Mining** process the information stored in the message.
- The service **Index Update** finally stores the document into the Index Store.
- While processing the data all framework components and services can use the Data Store for persisting their data.

**Search:**
- **Search Client** uses API to communicate with the framework.
- The Query processing is done within the **BPEL** engine.
- Finally, the BPEL service **Index Search** returns a search result back to the Search Client via API.

Figure 1: Architecture Overview

The architecture can be divided in two main functions:

- **Indexing** (left side of picture):
  - The Crawler crawls the data source and hands out the gathered data to the Connectivity module.
o The Connectivity Module normalizes incoming information to an internally used message format and pushes them into the Queue server. Large sets of incoming data could also be persisted into a data store to reduce the queue load.
o The BPEL engine listens to the queue and consumes the messages.
o BPEL services process the information with different services like Text-Mining, Rule engine and ontologies to produce annotations and annotated them in the message.
o The service „Index Update“ finally stores the document into the Index Store.
o While processing the data all framework components and services can use the Data Store to persist their data.

- Search (right side of picture):
o The Search Client uses an API to communicate with the framework.
o The Query processing is done within the BPEL engine.
o Finally the BPEL service Index Search returns a search result back to the Search Client via the API.

2.4 Related Technology

According to the description of the above architecture, SMILA will reuse third party software, mostly available under Open Source Licence.

- The OSGi specification about managing a component based software system (http://www.osgi.org/Main/HomePage).
- Equinox is a base technology from Eclipse implementing the OSGi specification (http://www.eclipse.org/equinox/).
- SCA - Service Component Architecture: An essential characteristic of a SOA is the ability to assemble new and existing services to create brand new applications that may consist of different technologies. The Service Component Architecture defines a simple, service-based model for construction, assembly and deployment of network of services (existing and new ones) that is language-neutral (http://www.oasis-open.org/committees/tc_cat.php?cat=soa). SCA is currently in the process of becoming an OASIS standard (http://www.oasis-open.org/committees/tc_cat.php?cat=soa) and is supported by many enterprise
software vendors (IBM, BEA, Oracle, SAP and more (see http://www.osoa.org/display/Main/Service+Component+Architecture+Partners).

- **Tuscany** (http://tuscany.apache.org/) is an Open Source SCA implementation. It seems Tuscany is the most complete and mature open source implementation currently. It is hosted at Apache and currently in incubation phase. Tuscany comes with a lot of 3rd party libraries itself:

3. Architectural Concepts of SMILA

This chapter describes essential concepts of the SMILA architecture.

3.1 Concepts for processing of records

In the following chapters the basic concepts for processing of records (which represents documents in most cases) are described:

- 3.1.1 ID Concept, as the basic data structure for identifying records
- 3.1.2 Structured ID object as the structured representation of the ID Concept
- 3.1.3 Record Data Model and XML representation, which contains further information about records identified by an ID.
- 3.1.4 Blackboard Service concept, which takes care of access and storage of records.
- 3.1.5 Router and Listener concept, which are access points to queues dispatching ID-related messages within the SMILA architecture.
- 3.1.6 BPEL Pipelining concept as the basic mechanism to orchestrate the services that process information contained in records.

3.1.1 ID Concept

The purpose of an ID is to identify an object in the system. An object in SMILA is:

- in a simple case a single document or
- a compound document that is an archive file (e.g. *.zip, *.chm -file) or big document that should be indexed by page or by section.

SMILA objects have a life cycle:

- Creation in a Crawler or an Agent
- Enrichment, splitting an merging (if it's possible) during processing in SMILA
- Persisting in storages (possibly in different states of processing) or indexes (usually at the end, but also possibly multiple times).
- Using the ID, it must be possible to refer to the source object.

The following definitions will be proposed:
- A **data source** is a single location providing access to a collection of data (Web server, file system, database, CMS, etc.). Data is read from a data source using crawler/agents. A data source must have a unique source ID within SMILA to refer to it without having to deal with the technical details of access.

- A **source object** is an entity in a data source. A Crawler or an Agent can create multiple SMILA objects from a single object source (e.g. by extracting files from a *.zip-file). A source object can be identified with respect to its data source using a relatively simple key (URL, path, primary key, etc.).

- A **record** is an entity representing a complete source object or a part of a source object to be processed by SMILA.
  - A record can be split into multiple records.
  - Multiple records referring to different parts of the same source object can be merged again. This can be useful to split large documents, process them section by section and merge the results again.
  - A record can be written to storages or indexes.
  - A record can be read from a storage in order to redo some of the processing (e.g. to rebuild an index after ontology changes).

- A **record ID**:
  - A record ID must contain and it must be able to extract the data source ID and the key of a source object in the data source, relative to the definitions of the data source.
  - A record ID must be provided by the Crawler or Agent.
  - Source objects can have multiple key values, e.g. in database tables with a primary key consisting of multiple columns.
  - During processing, the record ID can be enhanced by part specification after splitting a compound:
    - Element: part of a container, e.g. path in archive inducing recursion, attachment index in mails, etc. The element is identified by another key which is relative to the container element.
    - Fragment: identified by page number, section number, section name, etc.
  - If merging is supported, multiple records belonging to the same source object can be merged into a single record. The merged ID must reflect this.
  - According to these requirements a structured ID object will be used.
3.1.2 Structured ID object

This XML-nippet shows the elements of an ID object.

```
<smila:Record>
  <smila:ID>
    <smila:Source><![CDATA[String: ID of data source]]></smila:Source>
    <smila:Key><![CDATA[String: key of source object w.r.t. data source]]></smila:Key>
    <!-- the elements above are mandatory, the following is optional -->
    <smila:Element>
      <smila:Key><![CDATA[String: path in archive, attachment index]]></smila:Key>
      <!-- smila:Element can repeated for recursive archives -->
    </smila:Element>
    <smila:Fragment><![CDATA[page number, section name/number]]></smila:Fragment>
    <!-- maybe repeated e.g. for books: Part, Chapter, Section, Subsection ... -->
  </smila:ID>
  <!-- other metadata and non-binary content -->
</smila:Record>
```

For special cases like keys in a compound document more than one key element is necessary. For example a compound document from a database could have a primary key consisting of more than one column in the database schema. For a source object with multiple key values it must be distinguishable which key value belongs to which key "column". Therefore the element `<smila:Key>` can be optionally annotated with a name attribute like the next XML-Snippet.

```
<smila:Record>
  <smila:ID>
    <smila:Source><![CDATA[String: ID of data source]]></smila:Source>
```

The next example demonstrates the ID concept.

Assume a file system data source named "share", referring to a shared directory on a file server (e.g. "\fileserv\share"). It looks like this:

```
\fileserv\share
|-- PDF
| |-- big.pdf
| `-- Archive
|     |-- oldstuff.zip
|     |-- PDF
|         |-- old.pdf
|         `-- another.zip
|             `-- another.pdf
```

- "big.pdf" initially gets this ID:

```
<smila:ID>
  <smila:Source>share</smila:Source>
  <smila:Key>PDF/big.pdf</smila:Key>
</smila:ID>
```

- After splitting it by pages, the following ID refers to the first page of the document:

```
<smila:ID>
  <smila:Source>share</smila:Source>
  <smila:Key>PDF/big.pdf</smila:Key>
  <smila:Fragment>0</smila:Fragment>
</smila:ID>
```

- Similar for the ZIP: It starts as:

```
<smila:ID>
  <smila:Source>share</smila:Source>
  <smila:Key>Archive/oldstuff.zip</smila:Key>
</smila:ID>
```
When it is expanded, the contained file is referred to as:

```
<smila:ID>
  <smila:Source>share</smila:Source>
  <smila:Key>Archive/oldstuff.zip</smila:Key>
  <smila:Element>
    <smila:Key>PDF/old.pdf</smila:Key>
  </smila:Element>
</smila:ID>
```

Which in turn can be split into pages to become:

```
<smila:ID>
  <smila:Source>share</smila:Source>
  <smila:Key>Archive/oldstuff.zip</smila:Key>
  <smila:Element>
    <smila:Key>PDF/old.pdf</smila:Key>
  </smila:Element>
  <smila:Fragment>0</smila:Fragment>
</smila:ID>
```

And finally, the first page of the PDF in the recursive.zip would have this ID:

```
<smila:ID>
  <smila:Source>share</smila:Source>
  <smila:Key>Archive/oldstuff.zip</smila:Key>
  <smila:Element>
    <smila:Key>another.zip</smila:Key>
    <smila:Element>
      <smila:Key>another.zip</smila:Key>
    </smila:Element>
  </smila:Element>
  <smila:Fragment>0</smila:Fragment>
</smila:ID>
```

Similar, for a mail server as a data source "mail", we could have the following ID to refer to an attachment of a mail in folder INBOX. In this case, the Element name is the index of the Mime Message part in the message in this case:

```
<smila:ID>
  <smila:Source>mail</smila:Source>
  <smila:Key>INBOX/42</smila:Key>
  <smila:Element>
    <smila:Key>2</smila:Key>
  </smila:Element>
</smila:ID>
```
A row in a database table with a primary key consisting of columns x and y would be identified like this:

```xml
<smila:ID>
  <smila:Source>db</smila:Source>
  <smila:Key name="x">0815</smila:Key>
  <smila:Element>
    <smila:Key name="y">4711</smila:Key>
  </smila:Element>
</smila:ID>
```

3.1.3 Record Data Model and XML representation

The following requirements were detected:

- A simple API for service developers to work with the records.
- Minimal constraints on what is possible to express.
- Any SMILA component must be able to process every incoming record without knowing about any other component in the installation that may have produced some service specific part of the record. It must also be able to reproduce these elements in its result if they were not explicitly deleted during service execution. This means that for service specific classes we cannot even rely on having the same classes in the same version installed in each composite at the same time.
- Records produced and stored with one version state of a SMILA installation must be re-processable also with updated versions of the installation (at least, if the major version of the framework has not changed).
- XML representation
- Simple to express XPath queries on objects for conditions in BPEL or message routers.

Physical Data Model
Problems occur if different processing engines require different physical data models of the same logical object. E.g.: The ODE BPEL engine\(^7\) needs to be called with DOM objects and ActiveBPEL\(^8\) uses other classes. One could think of a SMILA specific processing engine that could use a physical data model that implements the logical data model more efficiently. Conversion between different physical models can become expensive if it has to be done very often. This means e.g. that if a BPEL engine to orchestrate a number of SMILA services, it should not be necessary to actually convert the exchanged data objects each time a service is called and each time a service returns its result to the engine. And because the orchestration engine should be replaceable like everything else in the framework, we cannot commit to using e.g. DOM as the physical representation of our data objects, because then we would have conversion issues when using ActiveBPEL. To solve this problem a Logical Data Model is designed.

**Description of proposed Logical Data Model**

A special requirement on a logical data model is to hide the physical implementation from the client in order to make optimized implementations of the data model possible in different parts of the framework.

Record, the top level element, can contain the following attributes (see also the example below):

- **ID**: see ID concept for details (s. chapter 3.1.1)
- **Metadata**: Metadata Object - the actual data about the document
- **Metadata Objects**: A list with Metadata Objects
- **Attachments**: additional data cannot be serializable to XML (or is too inefficient), e.g. binary content of documents or huge annotations
- **Attributes**: data about records according to some application or ontology models
  - Attribute with the sub-attributes:
    - o name: String
    - o value: List<MetadataObject|Literal>
    - o annotations: Map<String, List<Annotation>>

  - **Literal with the sub-attributes**:
    - o semantic type: String

---

\(^7\) [http://ode.apache.org/bpel-extensions.html](http://ode.apache.org/bpel-extensions.html)

\(^8\) [http://sourceforge.net/projects/activebpel](http://sourceforge.net/projects/activebpel)
3.1.3.1 Logical Data Model in XML

The following XML snippet illustrates how to possibly represent this data model in XML by example. The XML schema is targeted at being relatively easy to use for XPath expressions in BPEL processes or elsewhere. The element and attribute have been abbreviated in order to minimize the length on the resulting document. This should have a positive impact on communication overhead and processing performance.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- * Copyright (c) 2008 empolis GmbH. * All rights reserved. This program and the accompanying materials * are made available under the terms of the Eclipse Public License v1.0 * which accompanies this distribution, and is available at * http://www.eclipse.org/legal/epl-v10.html * * Contributors: * Juergen Schumacher (empolis GmbH) - initial example -->
<RecordList
 xmlns="http://www.eclipse.org/smila/record"
 xmlns:id="http://www.eclipse.org/smila/id"
 xmlns:rec="http://www.eclipse.org/smila/record"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.eclipse.org/smila/record.xsd">
 <Record version="1.0">
  <id:ID version="1.0">
   <id:Source>share</id:Source>
   <id:Key>some.pdf</id:Key>
  </id:ID>

  <A n="mimetype"> <!-- retrieval filter: annotation attached to attribute, valid for complete attribute value -->
   <An n="filter">
    <V n="type">exclude</V>
    <V n="values">
     <V>text/plain</V>
     <V>text/html</V>
    </V>
   </An>
  </A>
</RecordList>
```
Theseus-ORDO Deliverable D11.1.1.b

Version 1.0 Concept and Design of the Integration Framework Datum 25.09.2008

---

© Copyright by empolis GmbH

---

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html><head></head><body>

</body></html>
Some notes on this example:

- A `<RecordList>` has one record.
- The `<Record>` has an ID-Element like described above (see chapter 3.1.2).
- For example the `<Record>` has an attribute `<A>` named “topic”:
  - It has an Annotation-Object named „importance“ and a value `<V>”4.0“
  - It has a Literal `<L>` with a list of annotations `<an>` to the String “Eclipse”.
  - The annotations `<an>` have a name attribute “n” and one or more values.
- An attribute with a literal `<L>` contains multiple values `<V>`. So it’s possible to annotate an attribute. E.g. the sub-attribute “Eclipse” of attribute “topic” has an attribute “importance” with the value “2.0”.

© Copyright by empolis GmbH
• The "st"-attribute in a literal <L> and an object <O> means an application specific "semantic" type.

3.1.4 Blackboard Service concept

Purpose of the Blackboard Service is the management of SMILA record data during processing in a SMILA component (e.g. Connectivity (see chapter 3.2), Workflow Processor). The problem is that different processing engines could require different physical formats of the record data, hence, either complex implementations of the logical data model or data conversion problems is required.

The idea is to keep the complete record data only on a "blackboard" which is not pushed through the workflow engine itself and to extract only a small "workflow object" from the blackboard to feed the workflow engine. This workflow object would contain only the part from the complete record data which the workflow engine needs for loop or branch conditions (and the record ID, of course). Thus it could be efficient enough to do the conversion between blackboard and workflow object before and after each workflow service invocation. As a side effect, the blackboard service could hide the handling of record persistence from the services to make service development easier.

The following figure illustrates the data and message flows.
Workflow Processing using Blackboard Service

Note that the use of the Blackboard Service is not restricted to workflow processing, but it can be also used in Connectivity to create the initial SMILA record from the data sent by Crawlers. This way the persistence services are hidden from Connectivity, too.

It is assumed that the workflow engine itself (which will be a third party product usually) must be embedded into SMILA using some wrapper that translates incoming calls to workflow specific objects and service invocations from the workflow into real SMILA service calls. At least with a BPEL engine like ODE\(^9\) it must be done this way. In the following this wrapper is called the Workflow Integration Service. This Workflow Integration Service will also handle the necessary interaction between workflow engine and blackboard (see next section for details).

For ODE, the use of Tuscany SCA Java would simplify the development of this integration service because it could be based on the BPEL implementation type of Tuscany. However, in

the first version we will create a SMILA specific Workflow Integration Service for ODE that can only orchestrate SMILA pipelets because the Tuscany BPEL implementation type does not yet support service references [Tuscany Mailing list archive 2008]).

Recording to next figure below the next picture illustrates how and which data flows through this system.

Figure 3: Sequence diagram Workflow Integration and Blackboard Service

In more detail:

- Listener receives a record from queue. The record usually contains only the ID. In special cases it could optionally include some small attribute values or annotations that could be used to control routing inside the message broker.

- Listener calls blackboard to load record data from persistence service and writes attributes contained in message record to blackboard.

- Listener calls workflow service with ID from message record:
  - Workflow Integration Service creates a workflow object for given ID.
The workflow object uses engine specific classes (e.g. DOM for ODE BPEL engine) to represent the record ID and some chosen attributes that are needed in the engine for condition testing or computation. It's a configuration option of the workflow integration which attributes are to be included. In a more advanced version it may be possible to analyze the workflow definition (e.g. the BPEL process) to determine which attributes are needed.

- Workflow integration invokes the workflow engine. This causes the following steps to be executed a couple of times:
  - Workflow engine invokes SMILA service (pipelet). At least for ODE BPEL this means that the engine calls the integration layer which in turn routes the request to the invoked pipelet. So the workflow integration layer receives (potentially modified) workflow objects.
  - Workflow integration writes workflow objects to blackboard and creates record IDs. The selected pipelet is called with these IDs.
  - Pipelet processes IDs and manipulates blackboard content. The result is a new list of record IDs (usually identical to the argument list, and usually the list has length 1).
  - Workflow integration creates new workflow objects from the result IDs and blackboard content and feeds them back to the workflow engine.

- Workflow engine finishes successfully and returns a list of workflow objects.
  - If it finishes with an exception, instead of the following the Listener/Router has to invalidate the blackboard for all IDs related to the workflow such that they are not committed back to the storages, and also it has to signal the message broker that the received message has not been processed successfully such that the message broker can move it to the dead letter queue.

- Workflow integration extracts IDs from workflow objects and returns them.
- Router creates outgoing messages with message records depending on blackboard content for given IDs.
  - Two things may need configuration here: When to create an outgoing message to which queue (never, always, depending on conditions of attribute values or annotations) - this could also be done in workflow by setting a "nextDestination" annotation for each record ID. And which attributes/annotations are to be included in the message record - if any.
- Router commits IDs on blackboard. This writes the blackboard content to the persistence services and invalidates the blackboard content for these IDs.
• Router sends outgoing messages to message broker.

Content of the Blackboard

The Blackboard contains two kinds of content records and notes.

Records: All records currently processed in this runtime process. The structure of a record is defined in Data Model and XML representation (see chapter 3.1.3). Clients manipulate the records through Blackboard API methods. This way the records are completely under control of the Blackboard which may be used in advanced versions for optimised communication with the persistence services.

Records enter the blackboard by one of the following operations:

• create: create a new record with a given ID. No data is loaded from persistence, if a record with this ID exists already in the storages it will be overwritten when the created record is committed. E.g. used by Connectivity to initialize the record from incoming data.

• load: loads record data for the given ID from persistence (or prepare it to be loaded). Used by a client to indicate that it wants to process this record.

• split: creates a fragment of a given record, i.e. the record content is copied to a new ID derived from the given by adding a fragment name (see ID Concept for details).

All these methods should care about locking the record ID in the storages such that no second runtime process can try to manipulate the same record. A record is removed from the blackboard with one of these operations:

• commit: all changes are written to the storages before the record is removed. The record is unlocked in the database.

• invalidate: the record is removed from the blackboard. The record is unlocked in the database. If the record was created new (not overwritten) on this blackboard it should be removed from the storage completely.

Notes: Additional temporary data created by pipelets to be used in later pipelets in the same workflow, but not to be persisted in the storages. Notes can be either global or record specific (associated to a record ID). Record specific notes are copied on record splits and removed when the associated record is removed from the blackboard. In any case a note has a name and the value can be of any serializable Java class such that they can be accessed from separated services in own Virtual Machines (VM).
Pipelets running in a separate VM

**Blackboard Service: Processor with separated service**

Pipelets should be able to run in separated VM if they are known to be unstable or non-terminating in error conditions. This will be added in advanced versions of SMILA and discussed in more detail then.

So far the separated pipelet VM would have a proxy blackboard service that coordinates the communication with the master blackboard in the workflow processor VM. Only the record ID needs to be sent to the separated pipelets. However, the separated pipelet must be wrapped to provide control of the record life cycle on the proxy blackboard, especially because the changes done in the remote blackboard must be committed back to the master blackboard when the separated pipelet has finished successfully, or the proxy blackboard content must be invalidated without commit in case of an pipelet error. Possibly, this pipelet wrapper can also provide "watchdog" functionality to monitor the separated pipelet and terminate and restart it in case of endless loops or excessive memory consumption.
3.1.5 Router and Listener Concept

The Router (e.g. in Connectivity Module (chapter 3.2) normalizing incoming data to an internally used message format) processes crawled records and files them into a queue. The Router needs a configuration which contains rules about how records should be processed. The Router analyzes each record based on the rules and puts them regarding the rules into a queue.

Each DFP (Data Flow Process) contains a Listener which takes objects from a queue. A Listener reads a configuration that describes to which queue it should listen and which objects it should retrieve.

Following both configuration files are described. Both files use XML, thus each OSGi-Bundle (Router / Listener) contains a XML-Schema which describes how the configuration can be "used".

Router Configuration Description (see also the next figure):

- Rules contain the routing rules.
- DataSourceID Rule is for the first implementation, each record is routed based on the DataSourceID to a queue.
- Rule contains four Attributes:
  - WorkflowName (optional)
  - TargetQueue describes to which queue the record will be sent if the rule applies.
  - Operation.
  - Content
  - Each record is annotated with JMS-properties. Each property has a name and a value. These properties are used by the Listeners to decide which objects are to be retrieved from the queue.
Listener Configuration Description has the following attributes:

- Rules with the attributes:
  - SourceQueue and TargetQueue (optional). The Listener "connects" to this SourceQueue and is listening for objects in the queue that coincide with the properties (MessageSelector) of the rule (JMS-Properties).
  - Operation.
  - WorkflowName (optional).

### 3.1.5.1 Record Filter Concept

Record filtering can be useful in different parts of the system:

- the Queue Router needs it to create minimized objects to put in queue messages - see Router & Listener Queue Specification (s. chapter 3.1.5)
- the BPEL integration needs it to create workflow objects - see Blackboard Service Concept for details (s. chapter 3.1.4)

Therefore it should be provided as a generic functionality of the data model. Then we can

- provide a set of named record filter definition in a central place.
- refer to the names of record filters to be used in router and workflow engine configurations.
- both use the common code to actually do the filtering.

An initial record filter definition could consist of the following parts:
• name: unique name of the filter for reference in using components (Router or workflow engine configuration)
• list of attribute names: attributes to be kept in the filtered object. Additionally a flag could determine if annotations are to be copied, too. In an initial implementation it would be sufficient to have only top-level attributes here which means that the whole attribute tree with this name would be copied to the filtered object. It could be extended later to support attribute paths to specify filtering of sub-objects only.
• list of annotation names: names of top level annotations of the record to be kept in filtered objects.
• Example:

```xml
<RecordFilters>
  <Filter name="example">
    <Attribute name="Mimetype"/>
    <Attribute name="Filesize"/>
    <Attribute name="Keywords" keepAnnotations="true"/>
    <!-- default is false -->
    <Annotation name="action"/>
  </Filter>
  <!-- more filters -->
</RecordFilters>
```

### 3.1.6 BPEL Pipelining Concept

In this model the orchestration of pipelets is defined by BPEL processes. We distinguish two separate kinds of pipelets:

- "Big Pipelets" are implemented as OSGi services. They can be shared by multiple pipelines and their configurations are separated from the BPEL process definition.

- "Simple Pipelets" are managed by a component of the BPEL engine integration, instances are not shared by multiple pipelines and their configuration is part of the BPEL process definition.

In the following we assume that the service lifecycle of all services is controlled by OSGi Declarative Services (DS). This simplifies the starting and stopping of services and binding them to other services. To support the initialization of services at service activation, DS define that a special method is called when the service is activated, in which the necessary initialization can be done (reading of configurations, connecting to used resources, creating
internal structures, etc). DS also defines a method to be called when a service is deactivated that can be used for cleaning up. The two methods must have this signature:

- protected void activate(ComponentContext context);
- protected void deactivate(ComponentContext context);

Each pipelet service must have a service property "SMILA.pipelet.name" that specifies the name of this pipelet. The name must be unique for each service in a single VM and is defined in the DS component description. The pipelet name is used in BPEL definition to refer to the pipelets. If multiple instances of the same pipelet class are needed, they can be distinguished using different pipelet names.

The pipelet execution method is currently:

- Id[] process(Id[] recordIds) throws ProcessingException;

I.e. it is called by the workflow with a list of record IDs, the content of these records is supposed to be available via the Blackboard service, so all access and manipulation of the records is done using the Blackboard service. The result is also a list of record IDs. Usually these will be the same as the input IDs, a different list can be produced by pipelets that split records. This means that all data needed by the pipelet for processing must be on the blackboard:

- record attributes and attachments
- record annotations
- workflow and record notes

The two latter items may also be used to pass parameters to a pipelet. However, we will need BPEL Extension Activities to be able to set them in the BPEL definition (see end of this chapter).

Pipelets as well as the BPEL integration get their configurations from a central "configuration repository". This can be a simple directory with a defined structure at first, or a complex service supporting centralized configuration management and updating (and notification of clients about configuration changes) later.

Pipelet configurations are separated from the BPEL pipelines, because a Pipelet's existence does not depend on the existence of a pipeline engine and must not depend on the implementation of the pipeline engine. This makes it easier to use pipelets independent from a special pipelining implementation, e.g. if we want to replace the BPEL engine by a JBPM engine or an own workflow engine implementation. This makes it also easier to share pipelet instances between pipelines which is crucial for pipelets that use lots of memory (e.g.
semantic text mining) or need resources that can only be accessed exclusively by one client (e.g. writing to a Lucene index). Finally it enables OSGi to restart the BPEL integration service without having to restart the pipelets (e.g. for software updates).

The BPEL integration is started by DS, too. Pipelets are bound to the BPEL integration as DS service references. This way the BPEL service can always keep track about currently available pipelet services. It would even be possible to track which pipelet is used in which pipeline and thus to know a priori which pipeline is currently completely executable.

**Pipelet instantiation variants**

Usually we have one instance of a pipelet class that has a single configuration. The pipelet name is then like a key to the combination "pipelet instance name = pipelet class + configuration". However, there may be cases in which it would be good to have a single pipelet class available with different configurations. There are two ways to support this:

- Have a single pipelet instance with a configuration consisting of the different parts. Which part of the configuration is actually used in an invocation must then be passed using a record annotation. E.g.: There is a service "pipelet-name" = pipelet.A + configuration X & configuration Y, i.e. it has loaded both configurations.

- Have multiple pipelet instances with different names, each having one of these configurations. E.g. there are two service instances of the same pipelet class with different pipelet names:
  - service 1: "pipelet-name-1" = pipelet.B + configuration X
  - service 2: "pipelet-name-2" = pipelet.B + configuration Y

Then the pipelet name used in the BPEL invoked activity determines which configuration is used.

**Pipelet Implementation rules**

Pipelets can potentially be invoked more than once at the same time. This means that a pipelet either should be written in a multithreading-safe way (stateless, read-only configuration and member variables) or it must care itself about synchronization of critical sections (e.g. Lucene index writing).
3.2 Connectivity Module

The Connectivity Module is the entry point for external data. It is a single point of entry - on information level. The Connectivity Module normalizes incoming information to an internally used message format. Large sets of incoming data (binary data) should also be persisted into an external storage to reduce the queue load. It also includes functionality for buffering and routing of the incoming information. Its functionality is divided into several Sub-Components for better modularization. The Connectivity Module and its Sub-Components should all be implemented in Java. The external interfaces should also support SCA (see Glossary).

The next chart shows the Connectivity Module, its Sub-Components and their relationship as well as the relationship to other components:

![Connectivity Module Diagram](image-url)
Application Programming Interfaces (APIs)

Probably the Connectivity Module has to provide more than one interface/technologies for access. The main interface is used by the Information Reference Model (IRM) [see chapter 3.3] to provide crawled data objects. But it may also be used from within BPEL processes or from the Publish/Subscribe Module. These concepts focus on the interfaces used by IRM.

Processor

The Processor is the core of the Connectivity Module; it does the actual processing of the incoming data objects. The incoming data is stored depending on its type:

- Large or binary data is stored in a binary store (e.g. distributed file system).
- All other data are stored in a XML store (e.g. XML database).

The Processor also creates the message object to be queued. A message object contains the unique ID of the object, the Delta Indexing hash, routing information and any additional needed information. It should be configurable what information is part of a message.

The Processor should also be able to standardize incoming objects (either Records and/or message objects of the 2nd alternative interface design) to the latest version (internal representation) or to reject them.

Buffer

The Buffer delays the queue of outgoing messages. Therefore it needs a separate Queue mechanism to temporarily store the messages. This has not to be mistaken with the Queue Servers! The Buffer provides functionality to detect and resolve competing messages (add/update and delete of the same document).

Router

The Router routes messages to according Queues and/or BPEL workflows. The routing information (what whereto) has to be provided by the configuration. The Router also has to update the Delta Indexing information (see below) accordingly.

The only feedback the Router (and so the Connectivity Module) gets is if a message was queued or not. Therefore after a message was successfully queued one of the following actions must be triggered by the Router:

- add: create the Delta Indexing entry and mark as processed (visited)
- update: update the Delta Indexing entry and mark as processed (visited)
- delete: remove the Delta Indexing entry
It may be necessary to access directly the Router after a BPEL workflow has finished route a message to another Queue and therefore expand the API.

**Delta Indexing Manager**

The Delta Indexing Manager stores information about the last modifications of each document (even compound elements) and can determine if a document has changed. The information about the last modification should be some kind of Hash computed by the Crawler (see below IRM for further information chapter 3.3). It provides functionality to manage this information, to determine if documents have changed, to mark documents that have not changed (visited flag) and to determine documents that are indexed but no longer exist in the data source. The Delta Indexing Manager was moved inside the Connectivity Module for these reasons:

- some of it's functionality is used within the Connectivity Module
- as a single point of access the Connectivity Module should "know" about the delta indexing information
- In a distributed system we only need one connection from an IRM to the Connectivity Module and not a second one to access Delta Indexing Manager (this seems not to be a big gain, but may prove valid in high volume distributed scenarios).

Despite of being a part of the Connectivity Module, the implementation of Delta Indexing Manager is still replaceable to provide different stores for the delta indexing information (e.g. database or even a search index).

Here is a list of the information that needs to be stored by the Delta Indexing Manager:

- ID: the id of the document
- Hash: the hash of the document to determine modifications
- DataSourceID: the id of the data source from where the document was provided. This is already part of the document's ID, but we need it as separate value to clear by source
- IsCompound: flag, if the document is a compound object. This is needed to clean up recursively
- ParentID or ChildIDs: a reference to the parent document (if any exists) or references to child documents. This is needed to clean up recursively.
• VisitedFlag: flag that is temporary set during processing of a data source, to mark documents as visited. At the end all unmarked documents of a data source are deleted.

3.3 Information Reference Model (IRM)

The basic idea is to provide a framework to easily integrate data from external systems via Agents and Crawlers. The processing logic of the data is implemented once in so called Controllers, which use additional functionality provided by other components. To integrate a new external data source only a new Agent or Crawler has to be implemented. Implementations for Agents/Crawlers are not restricted to Java only. The technologies are based on Service Component Architecture (SCA)\(^\text{10}\) and Tuscany\(^\text{11}\).

The chart shows the architecture of the IRM framework with its pluggable components (Agents/Crawlers) and relationship to the SMILA entry point Connectivity Module.

\[^{10}\text{http://www.osoa.org/display/Main/Service\%2BComponent\%2BArchitecture\%2BSpecification}\]

\[^{11}\text{http://de.wikipedia.org/wiki/Apache_Tuscany}\]
Figure 7: Pluggable components in the IRM-Framework
The IRM Framework is provided and implemented by SMILA. Agents/Crawlers can be integrated easily by implementing the defined interfaces. An advanced implementation might support even both interfaces.

This below chart shows all components and their relationship on the SCA level:

- The green chevrons represent services provided by a component
- The purple chevrons represent references that the component relies on

**Agent Controller**

The Agent Controller implements the general processing logic common for all Agents. Its service interface is used by Agents to execute an add/update/delete-action.

**Agent**

Agents monitor a data source for changes (add/update/delete) or are triggered by events (e.g. trigger in databases).
Crawler Controller

The Crawler Controller implements the general processing logic common for all Crawlers. It has no service interface. All needed functionality should be addressed by a configuration/monitoring interface.

Crawler

A Crawler crawls actively a data source and provides access to collected data.

References

The Agent Controller and Crawler Controller have references to:

- Configuration Management: to get configurations for itself, Agents and Crawlers (see chapter 3.5.1)
- Connectivity Module (see chapter 3.2): as an entry point for the data for later processing by for example BPEL
- Compound Management handles processing of compound objects (e.g. *.zip-, *.chm-, *.rar-files).
- Delta Indexing Manager (see chapter 3.2). This is a Sub-Component of the Connectivity Module. Stores information about last modification of each document (even compound elements) and can determine if document has changed. The information about last modification could be some kind of HashToken. Each Crawler and the CompoundManagement should have its own configurable way of generating such a token. For file system it may be computed from last modification date and security information. For a database it may be computed over some columns. Some of its functionality is exposed through the Connectivity Module’s API.

3.4 XML Storage Service Concept

The XML Storage shall be used from several components (IRM, BPEL, Queue and Blackboard) within the SMILA. The main use case shall be to store and retrieve XML documents as well as to obtain a set of documents by an XPath/XQuery\(^\text{12}\).

---

The first API draft of SMILA shall define the basic „Create, Read, Update and Delete“-operations. In-place modifications of sub nodes are not yet needed.

It is suggested to publish the needed functionality as an OSGi Service with the possibility to multiple instances which may or may not be running in the same Virtual Machine. The latter case shall be covered by using SCA (see glossary) which handles this matter transparently to the user but imposes a few constraints in the API - at least in the Tuscany implementation\(^\text{13}\). These constraints are:

- Return values and parameters of methods must be serializable
- Overloading of methods is not allowed

**XML Storage Service**

The intended usage of the XML Storage is very complex. Hence the implementation shall be done as an OSGi Service that is wired up with Declarative Services\(^\text{14}\). The intended usage of the XML Storage is very much that of a service or server (e.g. like a real DB Server such as MySql, Oracle, etc.) as opposed to a library type implementation. Hence the implementation shall be done as an OSGi Service that is wired up with Declarative Services.

The service itself must support multiple requests at the same time and therefore needs to be multi-threaded. The intention is to use a connection-type approach as is the case for SQL databases. That entails that multiple clients may connect to the service and each client may open possibly multiple connections that are used to query/store XML documents concurrently.

An OSGi service is still run and called within the same Java Virtual Machine. This is in contrast to normal DB services that typically run in their own process and hence communication is done via TCP/IP, pipes etc. In the end we need to be able to access the Xml Storage Service remotely as well. This shall and can be done which SCA making services that matter transparent to the client and moving this aspect into configuration of the setup/installation.

Retrieval of a document may either be done by a String-key or formulating an XQuery which returns a sequence of XML nodes (types) and as such may return whole documents or part of a document.

\(^{13}\) [http://tuscany.apache.org/](http://tuscany.apache.org/)

Binary Storage

Although it is possible to save binary objects in Berkley DB XML and possibly other XML DBs it is better to provide separate OSGi Services for these distinctly different storage types. The possibilities for saving binary objects in XML databases are figured out later. First tests are detected the performance for larger binary objects is not good with Berkley DB.

3.5 Concepts of Infrastructure

This section describes three important concepts regarding the infrastructure of SMILA. Other concepts like Logging are currently in progress or are too specific, so that the reader would be overloaded with details; therefore it was avoided to explain it in this Deliverable.

3.5.1 Configuration Management

The configuration handling system should allow:

- To configure component groups (e.g. Agents/Crawlers).
- To configure single components (e.g. Log system).
- To configure using several sources (e.g. file system or GUI).
- Dynamically change and apply configurations.
- To configure distributed system.

The idea is to use the Configuration Admin Service (see below) as a basis and add an additional layer to adapt it to SMILA.

The Configuration Admin Service is an important aspect of the deployment of an OSGi Service Platform. It is a standard service in OSGi Service Platforms. It will be an implementation of the Configuration Admin Service of equinox\textsuperscript{15} (The Configuration Admin Service is described in [OSGi Compendium 2008, chapter 104].

It allows an operator to set the configuration information of deployed bundles. Configuration is the process of defining the configuration data of bundles and assuring that those bundles receive that data when they are active in the OSGi Service Platform.

\textsuperscript{15} http://www.eclipse.org/equinox/bundles/).
For SMILA it is suggested to add a ConfigurationAdminManager and a ConfigurationManagerRegistry to adopt base OSGi ConfigurationAdminService to SMILA. The OSGi Configuration Admin Service is used mainly for two purposes:

- It can be used as inner repository to store configuration (because configuration can be changed not only from file system).
- It is a standard OSGi configuration way it can be used in some cases (e.g. third party bundles).

It is possible to exchange the OSGi ConfigurationAdminService with another component without changing other parts of the configuration system.

### 3.5.2 Monitoring

SMILA needs functionality to provide:

- information about the state and availability of the whole system,
- information about the state and availability of single components,
- mechanisms to manage components (start, stop, restart, pause, resume, update, etc.).

Communication standards like Simple Network Management Protocol\(^\text{16}\) and JMX\(^\text{17}\) should be supported by the architecture (see next figure).


SNMP is seen as an add-on on top of JMX and is not a must have. Each component to be monitored therefore must provide an implementation of a so-called Agent. The Agent in Java components must support JMX. Additional SNMP functionality can be added using JMX. There are two possibilities (see the next figure):

- snmp4j\(^{18}\): an enterprise class free open source and state-of-the-art SNMP implementation for Java. It supports mapping from JMX MBean instrumentation to SNMP scalars, tables, and notifications. The coding has to be done manually.
- AdventNet SNMP Adaptor for JMX\(^{19}\): a SNMP to JMX adaptor that provides a configuration wizard for configuring the SNMP adaptor for user-defined MBeans\(^{20}\) and it automatically generates MIBs\(^{21}\). This is not Open Source.

\(^{18}\) [http://www.snmp4j.org/](http://www.snmp4j.org/)


\(^{20}\) [http://java.sun.com/j2se/1.5.0/docs/guide/management/overview.html#mbeans](http://java.sun.com/j2se/1.5.0/docs/guide/management/overview.html#mbeans)

\(^{21}\) [http://en.wikipedia.org/wiki/Management_Information_Base](http://en.wikipedia.org/wiki/Management_Information_Base)
Non Java Components (C++, .Net, etc.) do not support JMX. Depending on what protocol to support, there are two options (the first is the most flexible one):

- **JMX and/or SNMP**: to support JMX, a wrapping Java Agent has to be implemented. It the communication with the Non Java Component has to be implemented in the MBean classes, using some kind of communication protocol (JNI, Corba, etc.). SNMP functionality can be added to this Agent as for a regular Agent (see above).

- **SNMP only**: depending on the technology, they may directly support SNMP. For C++ there is a open source library Agent++ that could be used directly in a C++ component. So there is no need to implement any wrapping Java Agent.

**Monitoring Architecture Details**

![Monitoring Architecture](image)

Figure 10: Monitoring architecture in detail

### 3.5.3 Performance Measurement Framework

The goals of Performance Measurement Framework are:

- It has to deliver measurements of Application Metrics like response time, throughput, resource utilization, and workload.

- It has to be useable from each part of the distributed application and heterogeneous components (hardware / operation system)
Therefore, Performance Measurement Framework can be divided into two components:

- **Measurement Interface/API - Component**: Measurements are taken from application-specific results. Therefore the applications have to measure the metric itself. The results can be delivered to the Measurement interface/API which can be used from the Data Collection Component.

- **Data Collection Component**: The Data Collection Component can contact the Measurement Interface/API that is used in parts of the distributed application to collect the results of the measurement. Furthermore this component can analyze/convert the data and can create statistics or graphs.
4. **Conclusion**

This Deliverable described the fundamental concepts of SMILA, an Open Source framework for building semantic search applications. The development of this framework is driven by the requirements mentioned in the Chapter 2. From the authors’ point of view, all of these requirements are addressed by the concepts of SMILA.

Service oriented architectures (SOA) as fundamental paradigm of modern software architecture design is incorporated in the development of SMILA: The SOA-paradigm allows a comfortable integration of services from different vendors. Thereby all current and future-oriented standards like SCA, SDO, JMX, OSGi, WSDL and BPEL are applied.

With respect to these aspects SMILA has accepted the challenge for a modern integration platform for building an Information Access Management System.
5. Literature


- [Tuscany Mailing list archive 2008] Tuscany user mailing list (http://mail-archives.apache.org/mod_mbox/ws-tuscany-user/200804.mbox/%3c5a75db780804160846u6161d069p17c09a9422b2da8b@mail.gmail.com%3e) (last check 2008/07/04).
6. Glossary

A

- API – Application Programming Interface

B

- **BPEL** - is an XML-based language defining several constructs to write business processes. It defines a set of basic control structures like conditions or loops as well as elements to invoke web services and receive messages from services. It relies on WSDL to express web services interfaces. Message structures can be manipulated, assigning parts or the whole of them to variables that can in turn be used to send other messages.


D

- **Delta Indexing** - also known as incremental or generation based indexing.
- **DFP** - The Data Flow Process is a set of processing steps. These steps cover the following aspects and is described in the data flow process description:
  - Storage description - Extraction of messages from the queue
  - Process based information handling (e.g. splitting, routing, ...)
  - Data annotation through BPEL
- **DFPD** - The Data Flow Process Description is a set of process related configuration files. Files in this set are optional. The following components are contained in the DFPD:
  - Source/Target- references (e.g. Queue)
  - References to different storages or collections
  - BPEL (edit and delete process in several files organized in system/data processes)

E

- **Eclipse** - Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle ([http://www.eclipse.org/](http://www.eclipse.org/)).
- **Equinox** - a technology from Eclipse is a base technology implementing the OSGi specification. Not only delivering a high performance class loading
mechanism Equinox also provides an environment for managing component dependencies (http://www.eclipse.org/equinox/).

I

- **IRM** - Information Reference Model (see chapter 3.3)
- **IAM** – Information Access Management (see chapter 2.3)

O

- **ODE** - Apache ODE (Orchestration Director Engine) executes business processes written following the WS-BPEL standard. It talks to web services, sending and receiving messages, handling data manipulation and error recovery as described by your process definition. It supports both long and short living process executions to orchestrate all the services that are part of your application.
- **OSGi** - The OSGi specification is about managing a component based software system. It defines an in-VM Service Oriented Architecture (SOA) for networked systems. An OSGi Service Platform provides a standardized, component-oriented computing environment for cooperating networked services. This architecture significantly reduces the overall complexity of building, maintaining and deploying applications.

R

- **Record** - Sole element within SMILA data storage. A record may contain content and metadata.

S

- **SCA** - Service Component Architecture is a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture. SCA extends and complements prior approaches to implementing services, and SCA builds on open standards such as Web services. The SCA programming model is highly extensible and is language-neutral. Go to SCA and Tuscany for discussing (http://en.wikipedia.org/wiki/Service_component_architecture).
- **SDO** - Service Data Objects are designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and enterprise
information systems. The SDO programming model is language neutral (http://en.wikipedia.org/wiki/Service_Data_Objects).

- **SOA** - Service Oriented Architecture is a computer systems architectural style for creating and using business processes, packaged as services, throughout their lifecycle. SOA also defines and provisions the IT infrastructure to allow different applications to exchange data and participate in business processes. These functions are loosely coupled with the operating systems and programming languages underlying the applications (http://en.wikipedia.org/wiki/Service-oriented_architecture).

- **Tuscany - Apache Tuscany** is an implementation of the SCA specification 1.0. It is available for Java and C++. It also supports SDO specification 2.1 for both Java and C++. Go to SCA and Tuscany for discussing.

- **VM** – Virtual machine

- **WSDL** - WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate (http://en.wikipedia.org/wiki/Web_Services_Description_Language).

- **WS-BPEL** - see BPEL


- **XPath** - XPath (XML Path Language) is a language for selecting nodes from an XML document. (http://en.wikipedia.org/wiki/XPath)

- **XQuery** - XQuery is a query language (with some programming language features) that is designed to query collections of XML data. (http://en.wikipedia.org/wiki/XQuery)