Project no.: 257398

Co-funded by the
European Union

Project title: **Advanced predictive-analysis-based decision-support engine for logistics**

Start date of project: 01.10.2010 Duration: 36 months
End date of project: 30.09.2013
7^{th} FP topic addressed:
Challenge 4: Digital libraries and content: ICT-2009.4.3: Intelligent Information Management Objective
Small or medium scale focused research project (STREP)

D5.4: Developer Guide

Version 1.1

November 15, 2013

Contents

Executive summary

The document presents a guide to help software developers to understand the main ADVANCE open-source software frameworks and applications, namely the ADVANCE Flow Engine, the ADVANCE Flow Editor and the ADVANCE Live Reporter. The guide details the concepts, architecture and components in order to allow software developers to develop the frameworks and applications themselves or develop new solutions and plugins for these frameworks and applications.

1 Overview

The document describes the technical and development details of the ADVANCE project. Apart from the scientific results presented in other deliverables, this document provides details about the open-source softwares and libraries developed in the project:

- ADVANCE **Flow Engine** (section 2): a programming environment for data-stream processing.
- ADVANCE **Flow Editor** (section 3): visual editor for creating data-stream programs.
- ADVANCE **Elicitation Tool** (section 4): part of the cognitive modeller of WP8.
- ADVANCE **Live Reporter** (section 5): webapplication for real-time logistics tracking and prediction.

Readers of this document are expected to have sufficient technical knowledge about the *Java programming language*, especially version 7, the Extensible Markup Language (XML), object-oriented programming and relational databases.

In addition, readers are expected to be familiar with the following deliverables, which detail the concepts behind many components of the developed softwares (and not repeated here)

- **D3.3 Machine trackable flow descriptor format**: explains the concepts of the so-called flow description, the 'program' for the *Flow Engine*.
- **D3.4 Evolvable structured data-model and extension API**: explains the concepts behind the type system used by the *Flow Engine*.
- **D4.3 Statistical regression and generic optimization processing steps**: explains the theory behind the learning process implemented in the *Live Reporter*.
- **D5.1 Software requirements and test scenarios specification**: explains the design philosophy behind the softwares.
- **D5.2 Software solution and test results**: explains the high-lever workings of the software solution, suited for non-developers.
- **D6.1 User interface design**: explains the design concepts behind the *Flow Editor*.
- **D6.3 Modeller user interface**: explains more detailed the design concepts behind the *Flow Editor*.

The document lists source code, XML and other programming-related text. To help the reader visually distinguish between various programming text and the surrounding explanation text, programming-related texts are formatted with monospaced font and generally presented in blue:

```
<xml>Data</xml>
```

In general, XML and schema listings, names of XML elements and attributes, Java fields and method names are formatted this way.

In addition, the document lists several Java object names, which are distinguished by a different color:

- Java classes and abstract classes are formatted `in monospaced and slightly magenta color`.
- Java interfaces are formatted `in monospaced and green color`.
- Java enumerations are formatted `in monospaced and brown color`.

Such classes, interfaces and enums are also indexed and listed in the index section.

1.1 Abbreviations

The Guide refers to several abbreviations throughout the document. The table 1 lists all of the used abbreviations.

Abbrev.	Descriptions
ALR	*ADVANCE Live Reporter*, see section 5.
API	*Application Programming Interface*
FTP	*File Transfer Protocol*
J2EE	*Java 2 Enterprise Edition*
JDBC	*Java Database Connectivity*
JMS	*Java Messaging Service*
HTTP	*HyperText Transfer Protocol*
HTTPS	*Secure HTTP*
SMTP	*Simple Mail Transfer Protocol*
POP3	*Post Office Protocol version 3*
LGPL	*Lesser General Public License*
GPL	*General Public License*
CDDL	*Common Development and Distribution License*
SOAP	*Simple Object Access Protocol*
XML	*Extensible Markup Language*

Abbrev.	Descriptions
BLOB	*Binary Large Object*
CLOB	*Character Larger Object*

Table 1: Abbreviations used in the guide

2 ADVANCE Flow Engine

The ADVANCE Flow Engine is a data-flow-graph-based extensible software component which is based on reactive programming principles and utilizes various 3rd party libraries to handle the data-flow tasks and communicate with the outside world.

2.1 Architecture

Figure 1 shows an overview of the ADVANCE Flow Engine Architecture.

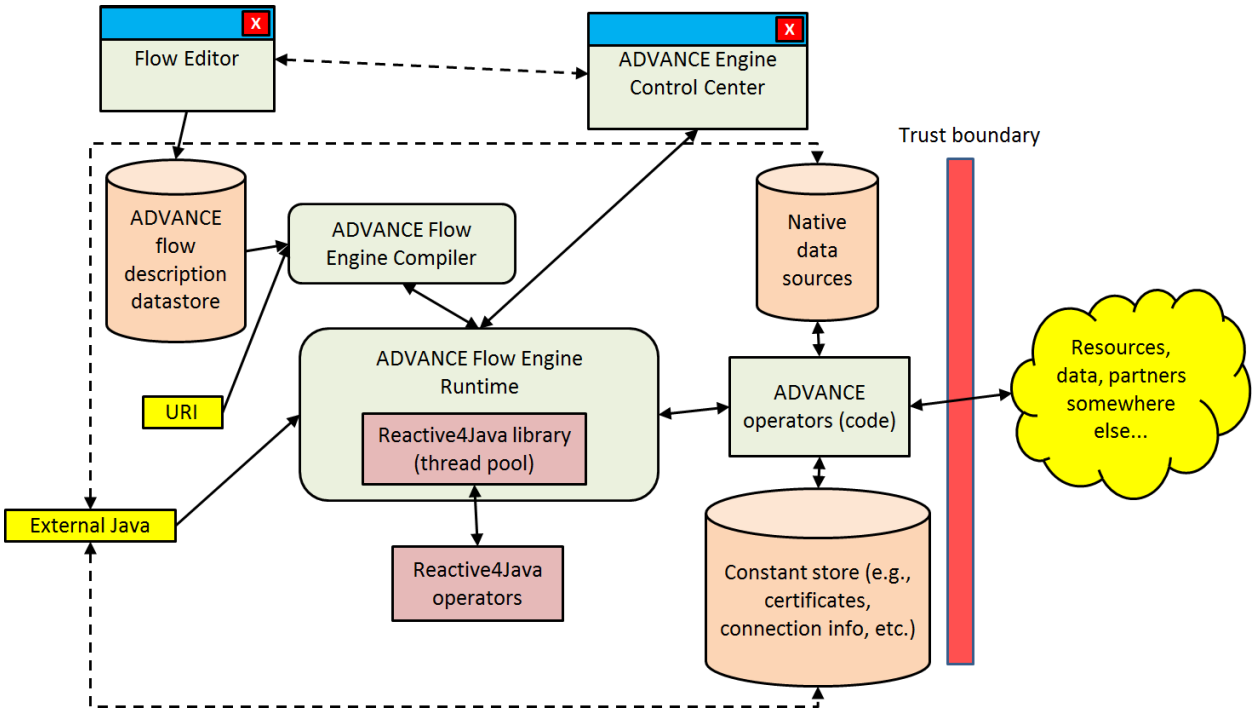

Figure 1: Flow Engine Architecture

Table 2 summarizes some properties of the listed components, most of which are described in the sections to follow.

Component	Properties
ADVANCE flow descriptor datastore	– Persistent storage for the XML definitions of the flow. – Might be independent or part of some existing or other datastore. – Could be a relational database or even a simple file system directory. – Is part of the Repository in the overall architecture.
URI	– Describes the location of any external XML flow descriptor – Probably local to company or server
External Java Library	– Contains third party operators, utility classes, etc., outside the scope of ADVANCE – Can be considered plugins – Might require/use the constant store or native resources – Is part of the Analytical Process Engine in the overall architecture
ADVANCE Flow Engine Compiler	– Transforms the XML flow descriptor format into a Java data model used by the runtime – Might emit data structures (expression trees) only, or – Might generate actual runnable Java code (which can then be subject to JIT optimizations from Java itself) – Is part of the Analytical Process Engine in the overall architecture
ADVANCE Flow Engine Runtime	– It hosts a thread pool where all code runs reactively or interactively – Interprets the compiled flow description and/or runs native Java code – Is part of the Analytical Process Engine in the overall architecture
Reactive4Java library & operators	– The components of the base framework – Gives strong guarantees about concurrency, sequencing, timing, etc. – Is part of the Analytical Process Engine in the overall architecture

Component	Properties
ADVANCE operators	– Developed by the ADVANCE project to support developing, running and testing the more sophisticated operations addressed by the project (learning, mining, scheduling, etc.) – Standard operators, GUI adaptors, etc. to support test applications or scenarios – Code fragments the Blocks are composed of – Part of the Analytical Process Engine in the overall architecture
Native data sources	– Legacy JDBC-reachable databases – Wrappers around PLCs (Programmable Logic Controllers), – Adaptors for internal web-services – Hand-held devices, etc. – Part of Data Store in the overall architecture
Constant store	– Storage for default values of blocks – Certificates for secure cross-company communication – Connection Strings to reach local JDBC databases – May be used for temporary data storage – Part of the Repository
External resources, data, partners, etc.	– Might represent an external ADVANCE APE – Web-services at other companies
Flow Editor	– Tool enabling the user to create, edit, describe and compose the flow description XMLs. – Supports testing, verification, import, export, etc.
ADVANCE Engine Control Center	– Controls the operation of the engine – Start/Stop, restart – Debug – Activate traces – Inspect internal data flows – May be integrated with the Flow Editor – May be a web page supplied by the Flow Engine – Create/maintain realms – Manage user access rights – Manage the Constant Store

Component	Properties

Table 2: Flow Engine Architecture components

2.2 Project organization

2.2.1 Directory structure

The project is organized into relatively flat directory structure:

Directory	Description
`src`	The source code directory.
`test`	Contains the JUnit test classes.
`conf`	Contains the default configuration files.
`schemas`	Contains the XSD files for the XML type system.
`lib`	Contains the 3rd party libraries.
`META-INF`	Contains the manifest files for the runnable applications.

Table 3: Flow Engine project directory structure

2.2.2 Package structure

The ADVANCE Flow Engine consists of several Java packages, organized by function.

(Since the package names are relatively long, table 4 contains the abbreviation `fe`, representing the package name `eu.advance.logistics.flow.engine`.)

Package	Description
`fe`	Main package containing the standalone engine, the compiler and the type system support classes.
`fe.api`	Contains the main API interface for the entire engine, compiler, XML-based communication and support record classes.
`fe.api.core`	Basic interfaces of the API and several basic exception classes.
`fe.api.ds`	The DataStore interface and basic record classes.
`fe.api.impl`	Engine access and DataStore implementations.
`fe.block`	Base class for ADVANCE blocks and data-transformations.

Package	Description
fe.block.*	Subpackages with ADVANCE block implementations.
fe.cc	The Control Center application's classes and dialogs.
fe.comm	File, HTTP, webservice, JMS, JDBC, Email accessor/communication classes.
fe.compiler	Compiler related classes.
fe.error	Compilation error classes.
fe.inference	The type inference related classes.
fe.model	The compilation error indicator interface.
fe.model.fd	The flow-descriptor related record classes.
fe.runtime	The general runtime block classes.
fe.test	Functional test classes (test directory).
fe.typesystem	The XML typesystem related classes.
fe.util	Few utility classes.

Table 4: Flow Engine project package structure

2.2.3 Libraries

Name	Ver.	License	Description
akarnokd-tools	-	Apache 2.0	General tools library from the developers of ALR. https://github.com/akarnokd/akarnokd-tools-and-utils
annotations	-	New BSD	JSR 305 Annotations. https://code.google.com/p/jsr-305
bcprov	1.4.9	MIT/Custom	Cryptography library. http://www.bouncycastle.org
commons-codec	1.6	Apache 2.0	Encoding/decoding library. http://commons.apache.org/proper/commons-codec/
commons-math3	3.2	Apache 2.0	Mathematical library. http://commons.apache.org/proper/commons-math/
commons-net	3.3	Apache 2.0	Network protocol library. http://commons.apache.org/proper/commons-net/
guava	14.0.1	Apache 2.0	Google collections library. https://code.google.com/p/guava-libraries/
httpclient	4.2.5	Apache 2.0	HTTP Client library. http://hc.apache.org/httpcomponents-client-ga/index.html
j2ssh	0.2.9	GPLv2	SSH/SFTP client library. https://sourceforge.net/projects/sshtools/
jaxen	1.1.6	Custom/OSS	XPath library. http://jaxen.codehaus.org/
javamail	1.4	CDDL/GPLv2	Email sender library. http://www.oracle.com/technetwork/java/javamail/index.html
jcommon	1.0.18	GLPv2	Custom GUI support classes. http://www.jfree.org/jcommon/
jms	1.1	Apache 2.0	Java Messaging API library. http://www.oracle.com/technetwork/java/jms/index.html
jsci	1.1	LGPL	Java Scientific library. http://jsci.sourceforge.net/
joda-time	2.2	Apache 2.0	Date and time library. http://joda-time.sourceforge.net/

Name	Ver.	License	Description
log4j	1.2.17	Apache 2.0	Logging library. http://logging.apache.org/log4j/1.2/
mysql-connector	5.1.25	GPLv2	MySQL database driver. http://dev.mysql.com/downloads/connector/j/
reactive4java	0.97	Apache 2.0	The reactive programming library. https://code.google.com/p/reactive4java/
slf4j	1.6.1	Apache 2.0	Simple logging facade to wrap all kinds of logging frameworks. http://www.slf4j.org/
trove	3.0.3	LGPL	Primitive collections library. http://trove.starlight-systems.com/

Table 5: Flow Engine libraries

2.3 Configuration

2.3.1 Flow engine configuration

By default, the Engine configuration is expected to be in the user's home directory under

`~/.advance-flow-engine-ws/LocalEngine/flow-engine-config.xml`

but configuration file and engine working directory can be specified via constructor parameters to the `AdvanceFlowEngine` class.

The configuration itself is managed by the `AdvanceEngineConfig` class. The configuration XML has the schema definition under the `schemas/flow-engine-config.xsd` or in the classpath root of the JAR file.

The structure of the configuration file is straightforward:

```
<flow-engine-config>
  <keystore/>
  ⋮
  <listener/>
  <datastore/>
  <scheduler/>
  ⋮
  <schemas>
  ⋮
  <block-registry/>
```

```
  ⋮
  <plugins></plugins>
</flow-engine-config>
```

2.3.1.1 keystore

The `keystore` elements define the *Java keystores* that contain public certificates ant private keys to be used by HTTPS inside the engine (e.g., for client/server authentication).

```
<keystore
  name='string'
  file='string'
  password='string'
/>
```

The `keystore` has a `name` attribute that lets other configuration elements refer to the keystore. The `file` contains the workdir-relative path to the keystore. The `password` contains the *Base64* obfuscated password to the keystore. The keystore settings are stored in `AdvanceKeyStore` records and in the `AdvanceEngineConfig.keystores` map.

2.3.1.2 listener

The `listener` defines the standalone Flow Engine's HTTPS-based access point.

```
<listener
  cert-auth-port='int'
  basic-auth-port='int'
  client-keystore='string'
  server-keystore='string'
  server-keyalias='string'
  server-password='string'
/>
```

The clients may connect to the `cert-auth-port` with client-certificate, or connect to the `basic-auth-port` with an username/password combination. The client's certificates are checked in the configured keystore referenced by the `client-keystore` attribute, the client's username/-password is checked through the datastore's *user table* described later on. The HTTPS server uses a X509 certificate on its own, which is stored in the keystore named by the `server-keystore` attribute and stored under a key name `server-keyalias`. Such private keys require additional password which is provided through the *Base64* obfuscated `server-password` attribute.

2.3.1.3 datastore

The DataStore, specified through the `datastore` element, is the primary storage place for the Engine's own data as well as configuration and resources provided to various blocks.

```
<datastore
  driver='string'
  url='string'
  user='string'
  password='string'
  schema='string'
  poolsize='int'
>
  <param name='string' value='any'/>
  ⋮
</datastore>
```

The DataStore, however, is not a general-purpose storage space. See section 2.4 for further details. The `datastore` contains a `driver` attribute, which determines whether a local, XML-based datastore (`"LOCAL"`) or a JDBC-compatible database is used (a proper JDBC driver class name, such as `"com.mysql.jdbc.Driver"`). The local datastore might be optionally encrypted via a *Base64* obfuscated password specified in `password` attribute. The encryption uses a password-based encryption with MD5 hash and DES cypher. The local datastore is managed by `LoadDataStore` class (2.4.2). In case the `datastore.driver` is not `LOCAL`, the JDBC connection information is specified via additonal attributes. The `url` attribute points to the database location in a driver-specific URL format. The `user` and the aforementioned `password` define the login credentials into the database. The `schema` specifies the the default database schema to use. The `poolsize` attribute limits the number of concurrent connections to the target database. The optional child elements `param` lets specify additional driver parameters (see table 8 for example).

2.3.1.4 scheduler

The `scheduler` elements contain configuration information for the main thread pools (instances of the `Scheduler` interface), identified by the `SchedulerPreference` enumeration, except for the `NOW` scheduler, which is always fixed and non-configurable.

```
<scheduler
  type='string'
  concurrency='string'
  priority='int/string'
/>
```

The `scheduler` element contains a `type` attribute, which contains the enumeration names of `SchedulerPreference` enum. The `concurrency` lets define the number of parallel threads of the scheduler, or the word `ALL_CORES` to use all available CPU cores. The `priority` attribute specifies the thread priority inside the scheduler. The content might be a number between 1 to 100 as a percentage, or the enumeration names of `SchedulerPriority`.

2.3.1.5 schemas

```
<schemas
  location='string'
/>
```

The optional `schemas` elements define a `location` attribute which is a working directory relative directory where the XSD schemas of the type system can be located. The Flow Engine stores expects the default type's XSDs in the classpath root.

2.3.1.6 block-registry

The optional `block-registry` elements

```
<block-registry
  file='string'
>
```

define a `file` attribute which is resolved against the working directory and contains the `block-registry.xsd` compatible block listing. See section 2.3.2 for further details. The Flow Engine contains a default set of blocks with a `/block-registry.xml` file in the classpath root.

2.3.1.7 plugins

The optional `plugins` element's content

```
<plugins>string</plugins>
```

overrides the default plugin directory `$workdir/plugins` to a different directory (still relative to the working directory). See section 2.11 for further details.

2.3.2 Block registry

The block registry (described by `/block-registry.xsd` schema) contains the definitions of the blocks runnable by the Flow Engine. Each block defines a Java class that extends the `Block` class, an unique identifier and the optional list of inputs and outputs. The Flow Engine supports multiple registry files defined either statically through the configuration (2.3.1) or by plugins dynamically. The `block-registry.xml` files are managed by the `BlockRegistryEntry` class.

Updating and maintaining such registry files is usually tedious, therefore, the Flow Engine offers Java annotations to specify the contents of a block registry entry, and an Annotation Processor library is supplied which automatically generates the `block-registry.xml` entries. See section 2.10.1 for further details.

The block registry has the following structure:

```
<block-registry>
  <block-description>
    <type-variable>
    ⋮
    <input/>
    ⋮
    <output/>
    ⋮
  </block-description>
  ⋮
</block-registry>
```

2.3.2.1 block-description

The `block-description` element contains several attributes.

```
<block-description
  id='string'
  class='string'
  scheduler='string'
  tooltip='string'
  keywords='string,string,...'
  category='string'
>
```

The class contains the fully qualified class name of the Block implementation. The id is a unique identifier of the block (unique in respect to all other blocks, either default or coming from a plugin). The scheduler attribute specifies the concurrency properties of the blocks. Its values are the names coming from SchedulerPreference enum. An optional tooltip attribute is used by the Flow Editor to display short information about the block. An optional keywords attribute specifies a comma separated list of keywords, potentially used to find similar blocks. The category attribute is used by the Flow Editor to group the blocks into a simple two level tree (see section 34 on the default category names). Each block-description element is managed by the AdvanceBlockDescription class.

2.3.2.2 type-variable

The Flow Engine supports generic typed blocks, in which case the type variable definitions have to be specified through the type-variable elements. It contains a name attribute which is then referenced from the other input and/or output elements. The documentation specifies a detailed description about the type variable. Type variable definition is managed by the AdvanceTypeVariable class.

```
<type-variable name='string' documentation='uri'>
  <upper-bound />
  <lower-bound />
</type-variable>
```

A type variable might define upper- and lower type bounds (although an upper bound usually doesn't make sense as the general block input behavior is covariant and the inputs are not allowed to be changed by the blocks). Both the upper-bound and lower-bound elements, exclusive to each other, are type definitions with the following structure:

2.3.2.3 type

The type structure is usually attached to some parent node and extends it with additional attributes and child nodes.

```
<[parent-node] type-variable='string' type='URI'>
  <type-argument/>
  ⋮
</[parent-node]>
```

The type-variable references a type-variable in the encompassing environment (such as a block definition). Exclusively, the type attribute defines the URI to the XSD defining the actual type structure. Default advance types are prefixed by advance: namespace (for example:

advance:string). Other prefixes are resolved from the containing XML's namespace definitions. For example:

```
<lower-bound
  xmlns:custom='http://custom.org/xsd'
  type='custom:type'
/>
```

The type-argument entries are themselves *type* nodes as well, but only apply when the parent type definition is not a type variable. A type definition is managed by the AdvanceType class.

2.3.2.4 input and output

Both input and output extend a type definition (2.3.2.3) with additional attributes which help identify, document and detail the input/output parameters.

```
<output
  id='string'
  displayname='string'
  documentation='string'
  [type-variable ...]
/>
```

Common attribute of input and output are the id attribute, specifying a block-unique identifier of the parameter. The displayname attributes allows the visual Flow Editor to display a custom name for the input/output port instead of the id value. The documentation is a free text displayed usually as tooltips when the user hovers over the input/output.

In addition, an input parameter definition may contain additional attributes:

```
<input
  id='' ...
  required='boolean'
  varargs='boolean'
>
  <default>
    <!-- XML element compatible with the input's type definition -->
  </default>
</input>
```

The required attribute specifies if the particular input parameter needs to be connected or is considered to be optional. The varargs attribute specifies if the given input may cover multiple, arbitrary number of input ports with the same name. It means when the block is being

placed by the Flow Editor, the user has to define the number of input arguments. Each of these variable number of arguments receive an unique name by appending the input's `id` with a running number (for example, `in1`, `in2`, ...). Unlike Java's variable argument support, a block may define several varargs-like parameters. See section 2.6.1 about how such blocks are defined in the flow description. The value of the `required` affects these attributes as a whole. Each input may define a default value (except the varargs typed ones), which is expected to be in the format the input's type definition requires.

2.4 Datastore

The ADVANCE Flow Engine datastore is a storage space backed by either file or relational database which stores several object types. Table 6 gives an overview of these object types and the associated Java class.

Object	Class	Description
Block state	`XNElement`	Raw, block specific state information in XML.
Email	`AdvanceEmailBox`	An SMTP/POP3 email box to collect emails from.
Flow descriptor	`XNElement`	Raw flow descriptor XML.
FTP	`AdvanceFTPDataSource`	FTP endpoint.
JDBC	`AdvanceJDBCDataSource`	JDBC endpoint.
JMS	`AdvanceJMSEndpoint`	JMS endpoint.
Keystore	`AdvanceKeyStore`	Java keystore for private keys and certificates.
Local files	`AdvanceLocalFileDataSource`	Reference to local files and directories.
Notification groups	-	Notification groups to contact users via various methods.
Realm	`AdvanceRealm`	Separate environments which run the flow programs.
SOAP	`AdvanceSOAPEndpoint`	Web service enpoint definition.
User	`AdvanceUser`	User preferences and access rights.
Web	`AdvanceWebDataSource`	HTTP(S) based web access.

Table 6: Datastore object types

2.4.1 API

To make access to the objects in table 6 independent to the underlying storage method, a public Java interface `AdvanceDataStore` is defined.

While selecting and deleting records use the same syntax across many database systems, insert-or-update an entry might be directly supported by SQL instructions. Therefore, methods updating the ADVANCE objects are placed into the separate `AdvanceDataStoreUpdate` interface. Database-specific implementations might be used by adding the fully qualified class name value to a `param` entry with a name `advance-ds-update-impl` into the Engine's configuration (2.3.1.3).

2.4.1.1 Exceptions

Each API method might throw an `java.io.IOException` or `AdvanceControlException`. The `IOException` refers to a communication exception or wraps the exceptions of other libraries and APIs (such as the `SQLException` from JDBC. This includes the exceptions thrown by the communication layers when using one of the remote, HTTP based APIs (e.g., the `HttpRemoteDataStore`). The `AdvanceControlException` is the functional exception of the method calls, potentially indicating access rights violations, invalid parameters, invalid system states, etc.

2.4.1.2 Access control

The DataStore API does not manage the access rights of the calling parties, i.e., both the reference `LocalDataStore` and `JDBCDataStore` implementations allow any methods to be called with any valid arguments. The access right checks are implemented through the `CheckedDataStore` wrapper class. The class takes an `AdvanceDataStore` instance and an user identifier, the *current user*. Once instantiated, the neither the datastore wrapped nor the user identifer can be changed. Usually, the `CheckedDataStore` is instantiated after a login process where the DataStore is first directly queried for the user's credentials for comparison.

User access rights are enumerated in the `AdvanceUserRights` Java enum. Each DataStore API has its own entry in the enum. The naming of the enums follows a simple convention:

`access mode _ object`

e.g., `LIST_USERS`, `CREATE_REALM`, etc.

Access to realms can be limited as well. The `AdvanceUserRealmRights` enumeration lists the access modes. Note that users need the `LIST_REALMS` right as well as the `LIST` realm right to actually see the particular realms. Table 7 lists the API methods and their required access rights.

Method	Rights required & Remarks
queryRealms	LIST_REALMS or MODIFY_USER, LIST – The MODIFY_USER is necessary so an andministrator can assign realm rights and therefore list the available realms. Lists only those realms where the user has the LIST right.
queryRealm	LIST_REALMS and LIST – The LIST is required for the target realm.
createRealm	CREATE_REALM
deleteRealm	DELETE_REALM
updateRealm	MODIFY_REALM
queryUsers	LIST_USERS
queryUser	LIST_USERS
enableUser	MODIFY_USER –
deleteUser	DELETE_USER –
updateUser	CREATE_USER or MODIFY_USER – Updating the current user is always allowed.
queryNotificationGroups	LIST_NOTIFICATION _GROUPS
queryNotificationGroup	LIST_NOTIFICATION _GROUPS
updateNotificationGroups	MODIFY_NOTIFICATION _GROUPS
queryJDBCDataSources	LIST_JDBC_DATA_SOURCES
queryJDBCDataSource	LIST_JDBC_DATA_SOURCES
updateJDBCDataSource	CREATE_JDBC_DATA_SOURCE or MODIFY_JDBC_DATA_SOURCE – Creating and updating is done via the same method.
deleteJDBCDataSource	DELETE_JDBC _DATA_SOURCE
queryJMSEndpoints	LIST_JMS_ENDPOINTS
queryJMSEndpoint	LIST_JMS_ENDPOINTS
updateJMSEndpoints	CREATE_JMS_ENDPOINT or MODIFY_JMS_ENDPOINT – Creating and updating is done via the same method.
deleteJMSEndpoint	DELETE_JMS _ENDPOINT
queryWebDataSources	LIST_WEB _DATA_SOURCES
queryWebDataSource	LIST_WEB _DATA_SOURCES

Table 7: CheckedDataStore methods access rights requirements

Method	Rights required & Remarks
updateWebDataSource	CREATE_WEB_DATA_SOURCE or MODIFY_WEB_DATA_SOURCE – Creating and updating is done via the same method.
deleteWebDataSource	DELETE_WEB_DATA_SOURCE
queryFTPDataSources	LIST_FTP_DATA_SOURCES
queryFTPDataSource	LIST_FTP_DATA_SOURCES
updateFTPDataSource	CREATE_FTP_DATA_SOURCE or MODIFY_FTP_DATA_SOURCE – Creating and updating is done via the same method.
deleteFTPDataSource	DELETE_FTP_DATA_SOURCE
queryLocalFileDataSources	LIST_LOCAL_FILE_DATA_SOURCES
queryLocalFileDataSource	LIST_LOCAL_FILE_DATA_SOURCES
updateLocalFileDataSource	CREATE_LOCAL_FILE_DATA_SOURCE or MODIFY_LOCAL_FILE_DATA_SOURCE – Creating and updating is done via the same method.
deleteLocalFileDataSource	DELETE_LOCAL_FILE_DATA_SOURCE
queryKeyStores	LIST_KEYSTORES
queryKeyStore	LIST_KEYSTORES
updateKeyStore	CREATE_KEYSTORE or MODIFY_KEYSTORE – Creating and updating is done via the same method.
deleteKeyStore	DELETE_KEYSTORE
querySOAPEndpoints	LIST_SOAP_ENDPOINTS
querySOAPEndpoint	LIST_SOAP_ENDPOINTS
updateSOAPEndpoint	CREATE_SOAP_ENDPOINT or MODIFY_SOAP_ENDPOINT – Creating and updating is done via the same method.
queryBlockState	DEBUG – Realm-specific right.
updateBlockState	DEBUG – Realm-specific right.
deleteBlockStates	DEBUG – Realm-specific right.
queryFlow	READ – Realm-specific right.
updateFlow	WRITE – Realm-specific right.
queryEmailBoxes	LIST_EMAIL
queryEmailBox	LIST_EMAIL

Table 7: CheckedDataStore methods access rights requirements

Method	Rights required & Remarks
updateEmailBox	CREATE_EMAIL or MODIFY_EMAIL – Creating and updating is done via the same method.
deleteEmailBox	DELETE_EMAIL

Table 7: CheckedDataStore methods access rights requirements

The method check() of the CheckedDataStore helps in checking the various user rights when the API methods are called. If the user lacks the expected rights, an AdvanceAccessDenied exception is thrown.

Most data store objects feature a *password* record field. In the default API, all field values are directly visible to clients in their raw form. However, the CheckedDataStore automatically strips the password information on any object passing through (clearPassword() method). Such objects are required to implement the HasPassword interface.

In addition, almost all DataStore objects implement the AdvanceCreateModifyInfo which stores when the particular object were added/modified and by which user. The classCheckedDataStore replaces the such object's modifiedBy fields with the user identifier used during the construction of the CheckedDataStore itself: object modifications will be automatically attributed to this user identifier (changeModifiedBy()). The contents of the createdAt and modifiedAt properties of the AdvanceCreateModifyInfo are usually ignored by the datastore implementation, and is replaced by the current time when the object is created or updated.

Some DataStore API methods take a byUser parameter in order to remember which user initiated the usually update-like methods. On the CheckedDataStore class, the parameter value is ignored and replaced by the current user. The reason behind this is that such methods require too few parameters to be defined with a record class, or the method only accesses some part of an existing object and needs to leave the other fields alone (for example, enabling an user account should not change anything but the enabled flag). The affected API methods are the following:

- createRealm()
- enabledUser()
- deleteUser()

2.4.2 Local-file DataStore

The local XML file-based DataStore implementation is in the LocalDataStore class.

Instantiating a LocalDataStore creates an empty datastore. Existing datastore XML files can be loaded via the load() or loadEncrypted() methods. The datastore has to be saved manually

with the save() or saveEncrypted() methods. When loading or saving as encrypted, the whole raw XML is encrypted/decrypted with a password-based encryption method (MD5 hash and DES cipher).

The class features HashMaps of the varios objects, usually keyed by their id() value (if they implement the Identifiable<T>). When objects are accessed through the DataStore API methods, each kind of object is syncrhonized { } over the respective map field.

Note that this local-file based datastore is there for demonstration purposes and lacks many reliability features a proper relational database holds.

2.4.3 JDBC DataStore

The relational database-based DataStore implementation is in the JDBCDataStore. It implements the AdvanceDataStore interface and uses a Pool<JDBCConnection> pool, that is used when connection to a database is required.

The JDBCDataStore accepts several configurationparameters (specified through the Engine's configuration file):

Parameter	Type	Description
advance-ds-update-impl	string	The fully qualified class name, implementing the AdvanceDataStoreUpdate interface, to be used.
advance-ds-xml-columns	boolean	Use a storage format where objects are stored as XML BLOBs and only their key is a separate column? See section 2.4.3.1 for more details.
advance-ds-update-mode	enum	Specifies how the insert-or-update methods should operate: see table 9 for valid values.

Table 8: JDBCDataStore configuration parameters

The DataStore API combines creating and updating objects into a single method call, which is then translated into one or multiple SQL queries. Depending on the underlying relational database's SQL dialect, several general and specific methods can be used which exhibit the so called *insert if not exist, update if exist* behavior. This mode can be set via the advance-ds-update-mode configuration parameter. The valid values are the names of the JDBCDataStoreUpdateMode enumeration.

Value	Description
SELECT_DECIDES	Execute an SQL SELECT and see if the object exists, then chose between INSERT and UPDATE.
INSERT_BEFORE_UPDATE	Execute an SQL INSERT and if fails with key violation, execute an UPDATE.
UPDATE_BEFORE_INSERT	Execute an SQL UPDATE and if the affected row number is zero, execute an INSERT.
MYSQL_REPLACE	Use the MySQL's REPLACE instruction.
ORACLE_MERGE	Use the Oracle's MERGE instruction.
MSSQL_MERGE	Use the MSSQL's MERGE instruction.

Table 9: Update mode values for the advance-ds-update-mode parameter

2.4.3.1 Database design

Each ADVANCE Flow Engine object from table 6 has its own database table. Depending on the setup, tables may contain the primary key and a XML BLOB field (indicated by an *italic* row), or all fields separately as columns.

In the following subsections, the table's primary keys are noted by underlining the relevant column names. The order in the primary key is the same as the order in the description tables.

The NUMBER type represents an integer-type, 64 bit length number. The VARCHAR type is a variable length string, usually required to be non-null (if null-ness is allowed, it is noted in the description column). The DECIMAL represents a floating point value, usually an IEEE-754 double precision value.

Name	Type	Description
realm	VARCHAR	The realm's name.
block_id	VARCHAR	The block's unique identifier.
state	BLOB	The state data as XML.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.

Table 10: ADVANCE_BLOCK_STATE table

Name	Type	Description
name	VARCHAR	The email box unique identifier.
receive	VARCHAR	Determines the email reception protocol. Valid values are the `AdvanceEmailReceiveProtocols` enum's names.
send	VARCHAR	Determines the email sending protocol. Valid values are the `AdvanceEmailSendProtocols` enum's names.
login	VARCHAR	The login mode into the email box. Valid values are the `AdvanceLoginType` enum's values.
sendAddress	VARCHAR	The server's internet address and port in the format `server[:port]` which sends out the emails. If the `port` is missing, the protocol's default port is used.
receiveAddress	VARCHAR	The server's internet address and port in the format `server[:port]` which receives and stores the incoming emails. If the `port` is missing, the protocol's default port is used.
folder	VARCHAR	The *inbox* folder name on the server.
email	VARCHAR	The sender's default email address.
keystore	VARCHAR	The keystore identifier in case the login type is `CERTIFICATE`.
user	VARCHAR	The user name for the login or the key alias for the certificate-based authentication.
password	VARCHAR	The plaintext (!) password for the login or the private key's password for the certificate-based authentication.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see `datastore-email.xsd` for the schema.*

Table 11: ADVANCE_EMAIL_BOX table

Name	Type	Description
realm	VARCHAR	The realm's unique identifier.
flow	BLOB	The flow XML, see `flow-description.xsd` for the schema.

Name	Type	Description
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.

Table 12: ADVANCE_FLOW table

Name	Type	Description
name	VARCAHAR	The FTP data source's unique name.
protocol	VARCHAR	The protocol, valid values are the `AdvanceFTPProtocols` enum's names.
address	VARCHAR	The FTP server's internet address in the format `address[:port]`.
remote_directory	VARCHAR	The default remote directory.
login	VARCHAR	The login mode into the FTP server. Valid values are the `AdvanceLoginType` enum's values.
user	VARCHAR	The user name or the key alias for a certificate-based login.
password	VARCHAR	The plaintext (!) password for the login or for the private key.
keystore	VARCHAR	The keystore name in case a certificate-based login is used.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see `datastore-ftp.xsd` for the schema.*

Table 13: ADVANCE_FTP table

Name	Type	Description
name	VARCHAR	The JDBC data source's unique identifier.
driver	VARCHAR	The JDBC driver's fully qualified class name.

Name	Type	Description
url	VARCHAR	The (driver specific) JDBC URL for the database connection.
user	VARCHAR	The user name for connecting to the database.
password	VARCHAR	The plaintext (!) password for connectiong to the database.
schema	VARCHAR	The default schema for the connection.
pool_size	NUMBER	The maximum number of concurrent connections allowed int the database.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see datastore-jdbc.xsd for the schema.*

Table 14: ADVANCE_JDBC table

Name	Type	Description
name	VARCHAR	The JDBC data source's unique identifier.
param_name	VARCHAR	The JDBC driver specific extra parameter name.
value	VARCHAR	The parameter value.

Table 15: ADVANCE_JDBC_PARAMS table

Name	Type	Description
name	VARCHAR	The JMS endpoint's unique identifier.
driver	VARCHAR	The JMS driver's fully qualified class name.
url	VARCHAR	The queue manager's URL.
user	VARCHAR	The login user for the queue manager.
password	VARCHAR	The plaintext (!) password for the queue manager.
queue_manager	VARCHAR	The queue manager name.
queue	VARCHAR	The send/receive queue name.
pool_size	NUMBER	The maximum number of connections to this endpoint.

Name	Type	Description
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see datastore-jms.xsd for the schema.*

Table 16: ADVANCE_JMS table

Name	Type	Description
name	VARCHAR	The keystore's unique identifier.
location_prefix	VARCHAR	The optional prefix to emulate a working directory.
location	VARCHAR	The keystore location.
password	VARCHAR	The plaintext (!) password.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see datastore-keystore.xsd for the schema.*

Table 17: ADVANCE_KEYSTORE table

Name	Type	Description
name	VARCHAR	The file's unique identifier.
directory	VARCHAR	The file or directory's path.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see datastore-local.xsd for the schema.*

Name	Type	Description

Table 18: ADVANCE_LOCAL_FILE table

Name	Type	Description
type	VARCHAR	The notification group type, valid values are the names of the `AdvanceNotificationGroupType` enum's items.
name	VARCHAR	The group's name.
value	VARCHAR	The notification value, e.g., email address, phone number, etc.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see `datastore-group.xsd` for the schema.*

Table 19: ADVANCE_NOTIFICATION_GROUP table

Name	Type	Description
name	VARCHAR	The realm's unique identifier.
status	VARCHAR	The realm's status, valid values are the names of the `AdvanceRealmStatus` enum's items.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see `datastore-realm.xsd` for the schema.*

Table 20: ADVANCE_REALM table

Name	Type	Description
name	VARCHAR	The SOAP endpoint's unique name.

Name	Type	Description
endpoint	VARCHAR	The endpoint's URL.
target_object	VARCHAR	The target object's URI.
target_namespace	VARCHAR	The target namespace's URI.
method	VARCHAR	The remote method name.
encrypted	NUMBER	Is the communication encrypted (`1 = true`).
keystore	VARCHAR	The keystore containing the key for encryption.
key_alias	VARCHAR	The name of the private key entry.
password	VARCHAR	The plaintext (!) password for the private key.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see `datastore-soap.xsd` for the schema.*

Table 21: ADVANCE_SOAP table

Name	Type	Description
name	VARCHAR	The user identifier.
right	VARCHAR	The user right, valid values are the names of the `AdvanceUserRights` enum's items.

Table 22: ADVANCE_USER_RIGHTS table

Name	Type	Description
name	VARCHAR	The user identifier.
realm	VARCHAR	The realm.
right	VARCHAR	The user right in this realm, valid values are the names of the `AdvanceUserRealmRights` enum's items.

Table 23: ADVANCE_USER_REALM_RIGHTS table

Name	Type	Description
name	VARCHAR	The user's unique identifier.
enabled	NUMBER	Is the user enabled (`1 = true`).
email	VARCHAR	The user's email address.
pager	VARCHAR	The user's pager number.
sms	VARCHAR	The user's SMS number.
date_format	VARCHAR	The date formatting preference of the user, see `DateFormat` for the patterns.
datetime_format	VARCHAR	The date and time formatting preference.
number_format	VARCHAR	The numer formatting preference, see `NumberFormat` for the patterns.
thousand_separator	VARCHAR	The thousand separator preference.
decimal_separator	VARCHAR	The decimal separator preference.
login	VARCHAR	The login type, see `AdvanceLoginType` enum names for valid values.
password	VARCHAR	The plaintext (!) password for the login or the key.
keystore	VARCHAR	The keystore name.
key_alias	VARCHAR	The key alias.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see `datastore-user.xsd` for the schema.*

Table 24: ADVANCE_USER table

Name	Type	Description
name	VARCHAR	The web data source unique name.
url	VARCHAR	The endpoint URL.
login	VARCHAR	The login type, see `AdvanceLoginType` enum's items.
keystore	VARCHAR	The keystore reference.
user	VARCHAR	The user name for login or the key alias name.

Name	Type	Description
password	VARCHAR	The plaintext (!) password for the login or the private key.
created_by	VARCHAR	The user's identifier who created this record.
created_at	DATETIME	The timestamp when the record was created.
modified_by	VARCHAR	The user's identifier who modified this record the last time.
modified_at	DATETIME	The last modification timestamp.
xml_data	*BLOB*	*The record as a single XML, see `datastore-web.xsd` for the schema.*

Table 25: ADVANCE_WEB table

2.5 Engine

2.5.1 API

The Flow Engine provides the interface `AdvanceEngineControl` to interact with it locally or remotely.

Each API method might throw a `java.io.IOException` or an `AdvanceControlException`. The `IOException` refers to a communication exception or wraps the exceptions of other libraries and APIs. This includes the exceptions thrown by the communication layers when using one of the remote, HTTP based APIs (e.g., the `HttpRemoteEngineControl`). The `AdvanceControlException` is the functional exception of the method calls, potentially indicating access rights violations, invalid parameters, invalid system states, etc.

The Engine API does not manage the access rights of the calling parties, i.e., both the reference `LocalEngineControl` and `HttpRemoteEngineControl` implementations allow any methods to be called with any valid arguments. The access right checks are implemented through the `CheckedEngineControl` wrapper class. The class takes an `AdvanceEngineControl` instance and an user identifier, the *current user*. Once instantiated, the neither the engine control wrapped nor the user identifer can be changed.

Table 26 lists the API methods and the required user rights.

Method	Rights	Remarks
`getUser`	-	Returns the *current user*.
`queryVersion`	-	Returns the Engine's version record `AdvanceEngineVersion`.

Table 26: Engine control method access rights.

Method	Rights	Remarks
queryBlocks	LIST_BLOCKS	Returns a list of block definitions (BlockRegistryEntry, see 2.3.2).
querySchemas	LIST_SCHEMAS	Returns a list of schema objects of AdvanceSchemaRegistryEntry.
querySchema	LIST_SCHEMAS	-
updateSchema	CREATE_SCHEMA or MODIFY_SCHEMA	Adds or changes a schema.
deleteSchema	DELETE_SCHEMA	-
deleteKeyEntry	DELETE_KEY	Delete a key from a keystore.
generateKey	GENERATE_KEY	Generate a key based on the AdvanceGenerateKey settings.
exportCertificate	EXPORT _CERTIFICATE	Export a certificate based on a AdvanceKeyStoreExport settings.
exportPrivateKey	EXPORT _PRIVATE_KEY	Export a private key based on a AdvanceKeyStoreExport settings.
importCertificate	IMPORT _CERTIFICATE	Import a certificate based on a AdvanceKeyStoreExport settings.
importPrivateKey	IMPORT _PRIVATE_KEY	Import a private key based on a AdvanceKeyStoreExport settings.
exportSigningRequest	EXPORT _CERTIFICATE	Create a certificate signing request.
importSigningResponse	IMPORT _CERTIFICATE	Import a signed certificate.
testJDBCDataSource	LIST_JDBC _DATA_SOURCES	Test if a given JDBC data source is reachable from the engine.
testJMSEndpoint	LIST _JMS_ENDPOINTS	Test if a given JMS endpoint is reachable from the Engine.
testFTPDataSource	LIST_FTP _DATA_SOURCES	Test the given FTP data source is reachable from the Engine.

Table 26: Engine control method access rights.

Method	Rights	Remarks
queryKeys	LIST_KEYS	Lists the keys of a keystore in AdvanceKeyEntry records.
startRealm	START	Start a specific realm.
stopRealm	STOP	Stop a specific realm.
queryFlow	READ	Query the flow description currently running in the specific realm as a AdvanceCompositeBlock object.
updateFlow	WRITE	Update a flow description in a realm, stopping, compiling and restarting it if necessary.
verifyFlow	LIST_BLOCKS and LIST_SCHEMAS	Verify a flow description.
debugBlock	DEBUG	Attach a debugger to a specific block in the specific realm and observe a sequence of BlockDiagnostic events.
debugParameter	DEBUG	Attack a debugger to a specific block's input or output port and observe a sequence of PortDiagnostic events.
injectValue	DEBUG	Send a custom value to a block's input port.
queryCompilationResult	READ	Returns the last compilation result (errors) of the specific realm.
queryPorts	IO	Returns the global input and output ports of a realm in a list of AdvancePortSpecification records.
receivePort	IO	Observe the raw messages of a specific output port.
sendPort	IO	Send a sequence of raw messages to a set of input ports of a specific realm.

Table 26: Engine control method access rights.

The method check() of the CheckedEngineControl helps in checking the various user rights when the API methods are called. If the user lacks the expected rights, an AdvanceAccessDenied exception is thrown.

2.5.1.1 Communication: client side

Communicating with a Flow Engine is possible through XML messages exchanged over a regular HTTPS connection.

From a client's perspective, the HttpRemoteEngineControl, implementing the AdvanceEngineControl, takes care of formulating the requrests and parsing the responses from a remote Flow Engine.

Internally, the HttpRemoteDataStore class, implementing the AdvanceDataStore, manages the messages towards the datastore returned by the datastore() method. It uses the same connection as the regular engine control messages.

Instantiating a HttpRemoteEngineControl, several overloads are available. Common parameter is the URL to the remote engine. Further parameter sets can be:

- an AdvanceHttpAuthentication record specifying a simple username/password or a certificate-based login,
- an username and password,
- an username, password and a Java KeyStore to verify the server's HTTPS certificate,
- a custom AdvanceXMLCommunicator instance, i.e., a shared connection.

Internally, the first 3 overloads instantiate a HttpCommunicator (implementing the AdvanceXMLCommunicator) that manages the proper SSL connection settings and converts its input XML data into character data and the answer back again.

An important property of such AdvanceXMLCommunicator is that it should be able to send and receive messages while one or more of its receive() methods are active. The HttpCommunicator provides this by making independent connections for each of the methods.

Transforming the Java objects of the Engine API is done via the XNSerializables helper class. Most API objects implement the XSerializable interface and are marshalled and unmarshalled in a straightforward manner. Table 27 lists the Engine API's request and response XML formats (schemas are located in the classpath root or in the schemas project directory).

Method	Request	Response
getUser	api-get-user.xsd	datastore-user.xsd

Table 27: Engine API XML message formats

Method	Request	Response
queryBlocks	api-query-blocks.xsd	api-blocks.xsd
querySchemas	api-query-schemas.xsd	api-schemas.xsd
queryVersion	api-query-version.xsd	api-version.xsd
updateSchema	api-update-schema.xsd	-
querySchema	api-query-schema.xsd	api-schema.xsd
deleteKeyEntry	api-delete-key.xsd	-
generateKey	api-generate-key.xsd	-
exportCertificate	api-export-cert.xsd	string.xsd
exportPrivateKey	api-export-key.xsd	string.xsd
importCertificate	api-import-cert.xsd	-
importPrivateKey	api-import-key.xsd	-
exportSigningRequest	api-export-signing.xsd	string.xsd
importSigningRequest	api-import-signing.xsd	-
testJDBCDataSource	api-test-jdbc.xsd	string.xsd
testJMSEndpoint	api-test-jms.xsd	string.xsd
testFTPDataSource	api-test-ftp.xsd	string.xsd
queryKeys	api-query-keys.xsd	api-keys.xsd
startRealm	api-start-realm.xsd	-
stopRealm	api-start-realm.xsd	-
queryFlow	api-query-flow.xsd	flow-description.xsd
updateFlow	api-update-flow.xsd	-
verifyFlow	api-verify-flow.xsd	api-compilation-result.xsd
debugBlock	api-debug-block.xsd	api-block-diagnostic.xsd
debugParameter	api-debug-param.xsd	api-port-diagnostic.xsd
injectValue	api-debug-inject.xsd	-
shutdown	api-shutdown.xsd	-
deleteSchema	api-delete-schema.xsd	-
queryCompilationResult	api-query-compilation-result.xsd	api-compilation-result.xsd
queryPorts	api-query-ports.xsd	api-ports.xsd
receivePort	api-receive-port.xsd	api-receive.xsd

Table 27: Engine API XML message formats

Method	Request	Response
sendPort	api-send-port.xsd	-

Table 27: Engine API XML message formats

The DataStore API's request and response XML formats are listed in table 28. In general, the response XML schemas match the LocalDataStore's and JDBCDataStore's XML record format.

Method	Request	Response
queryRealms	api-query-realms.xsd	api-realms.xsd
queryRealm	api-query-realm.xsd	datastore-realm.xsd
createRealm	api-create-realm.xsd	-
deleteRealm	api-delete-realm.xsd	-
updateRealm	api-update-realm.xsd	-
queryUsers	api-query-users.xsd	api-users.xsd
queryUser	api-query-user.xsd	datastore-user.xsd
queryNotificationGroups	api-query-groups.xsd	datastore-group.xsd
queryNotificationGroup	api-query-group.xsd	api-group.xsd
updateNotificationGroups	api-update-groups.xsd	-
queryJDBCDataSources	api-query-jdbc -ds.xsd	api-jdbc-ds.xsd
queryJDBCDataSource	api-query-jdbc.xsd	datastore-jdbc.xsd
updateJDBCDataSource	api-update-jdbc.xsd	-
deleteJDBCDataSource	api-delete-jdbc.xsd	-
queryJMSEndpoints	api-query-jms-es.xsd	api-jms-es.xsd
queryJMSEndpoint	api-query-jms.xsd	datastore-jms.xsd
updateJMSEndpoint	api-update-jms.xsd	-
deleteJMSEndpoint	api-delete-jms.xsd	-
queryWebDataSources	api-query-web-ds.xsd	api-web-ds.xsd
queryWebDataSource	api-query-web.xsd	datastore-web.xsd
updateWebDataSource	api-update-web.xsd	-
deleteWebDataSource	api-delete-web.xsd	-
queryFTPDataSources	api-query-ftp-ds.xsd	api-ftp-ds.xsd
queryFTPDataSource	api-query-ftp.xsd	datastore-ftp.xsd

Table 28: DataStore API XML message formats

Method	Request	Response
updateFTPDataSource	api-update-ftp.xsd	-
deleteFTPDataSource	api-delete-ftp.xsd	-
queryLocalFileDataSources	api-query-files.xsd	api-files.xsd
queryLocalFileDataSource	api-query-file-xsd	datastore-local.xsd
updateLocalFileDataSource	api-update-file.xsd	-
deleteLocalFileDataSource	api-delete-file.xsd	-
queryKeystores	api-query-keystores.xsd	api-keystores.xsd
queryKeystore	api-query-keystore.xsd	datastore-keystore.xsd
updateKeyStore	api-update-keystore.xsd	-
deleteKeyStore	api-delete-keystore.xsd	-
querySOAPEndpoints	api-query-soap-es.xsd	api-soap-es.xsd
querySOAPEndpoint	api-query-soap.xsd	datastores-soap.xsd
updateSOAPEndpoint	api-update-soap.xsd	-
deleteSOAPEndpoint	api-delete-soap.xsd	-
queryEmailBoxes	api-query-emails.xsd	api-emails.xsd
queryEmailBox	api-query-email.xsd	datastore-email.xsd
updateEmailBox	api-update-email.xsd	-
deleteEmailBox	api-delete-email.xsd	-
hasUserRight	api-has-user-right.xsd api-has-user-realm-right.xsd	boolean.xsd
queryBlockState	api-query-block-state.xsd	Block-dependent
updateBlockState	api-update-block-state.xsd	-
deleteBlockStates	api-delete-block-states.xsd	-
queryFlow	api-query-flow.xsd	flow-description.xsd

Table 28: DataStore API XML message formats

2.5.1.2 Communication: server side

On the receiving end of the Flow Engine, the `HttpEngineControlListener` and `HttpDataStoreListener` converts the incoming raw XML messages into proper `AdvanceEngineControl` and `AdvanceDataStore` method calls.

Both listeners are based on the `AdvanceHttpListener` interface, which declares a single `dispatch()` method. The method receives the raw XML message and the user name of the logged-in user. In general, the `dispatch` decides on which inteface method to call via the name of the root node of the raw XML message. In case the proper method couldn't be determined, an `AdvanceControlException` is raised.

2.5.1.3 Communication: multiple results

The result is a `AdvanceXMLExchange` instance. It manages none, single or multiple result values with the respectively named static methods. Internally, the instance manages a blocking `Iterable` sequence of values, which might actually wrap an asynchrnous `Observable<XNElement>` sequence of values by itself. In some scenarios, observing values might produce so much data that the server would run out of memory before it transmits the individual elements. To avoid this, the `AdvanceXMLExchange.multiple` flag indicates, that after each element, the underlying communication channel is flushed. Such response messages are wrapped into a `multiple-fragments` element. The client should parse the child nodes of this (XML fragments) as they appear. The client-side `HttpCommunicator` automatically performs this conversion. On the server side, the `AdvanceFlowEngine` main class' `createHandler()` method creates a HTTP responder that performs the chunking of the data. The very same purpose is established in the ADVANCE Live Reporter's `AdvanceFlowEngineServlet` (see 5.5.2).

2.5.1.4 Communication: authentication

In the reference implementation of the `AdvanceFlowEngine` class, users are authenticated when the HTTPS connection is established. In case of a simple basic authentication, the `setBasicAuthenticator()` method sets up Java's built-in HTTP server to check for the credentials. If the user is found in the datastore and the passwords match, the HTTP request object stores the user name under the `LOGIN_USERNAME` key. When a certificate-based login takes place (setup via `setCertAuthenticator()` method), the user's certificate are matched against the configured client-keystore's certificates. If a matching certificate is found, its key alias is used for finding the user in the ADVANCE DataStore. Such user needs the login type set to certificate-based or the login is refused.

2.6 Flow description

The *Flow Description* contains the 'program' for the Flow Engine to execute. It contains information about computation blocks and data connections between them and the environment.

2.6.1 Structure

The flow description is an XML file with a well-defined structure found in `flow-description.xsd` and consists of the following building blocks:

- `block` representing a computation logic.
- `port` representing the connection interfaces between blocks as inputs and outputs.
- `binding` or `wire` connecting the output ports of blocks to input ports of other blocks.
- `composite-block` which encloses other blocks and other composite blocks, forming a tree-like hierarchy.
- `constant` representing fixed values to be used for block inputs.

```
<flow-description>
  <composite-block>...</composite-block>
</flow-description>
```

The `flow-description` contains a `composite-block`, which is the root of the program. A composite block may contain type definitions, input and output ports, blocks, other composite-blocks and wires binding all of them.

```
<composite-block
    id='string'
    documentation='string'
    keywords='string,string,...'
    x='int'
    y='int'
>
  <type-variable>...</type-variable>
  ⋮
  <input>...</input>
  ⋮
  <output>...</output>
```

```
  ⋮
  <block>...</block>
  ⋮
  <composite-block>...</composite-block>
  ⋮
  <bind>...</bind>
  ⋮
  <constant>...</constant>
  ⋮
</composite-block>
```

Each composite block contains a flow-global unique identifier (`id`), documentation and keywords attributes, and the on-screen location `x` and `y` used by the Flow Editor to remember the visual layout of a composite block.

Each composite block may define its own set of type variables to support generic but type-safe data processing. See 2.3.2.2 for its structure.

Composite blocks may have `input` and `output` ports defined, which lets blocks and wires connect to it. Both may have generic types. Accessing these ports on a compiled flow program is not possible, except for the outermost composite block where the `AdvanceEngineControl` API's `sendPort()` and `receivePort()` methods can be used. Their structure is based on the block registry's input/output elements (see 2.3.2.4) with the exception of having an optional `keywords='string,string,...'` attribute. The composite block's input and output parameters provide a dual view. From the outside, they behave as any regular block's input and output, the former takes data and the latter produces data. However, inside the composite blocks, these very same parameters perform effectively the opposite: input parameters provide data and output parameters take data. Regular blocks inputs can be bound to these input parameters.

The `block` elements are the instantiations of the computation blocks of the Flow Engine.

```
<block
    id='string'
    type='string'
    documentation='string'
    keywords='string,string,...'
    x='int'
    y='int'
>
  <vararg name='string' count='int'/>
  ⋮
</block>
```

Besides the typical documentation and keywords attributes, the `id` attribute is a flow-global unique identifier of this block instance. The `type` references the block name in the block registry (see 2.3.2). Attributes `x` and `y` is used by the Flow Editor to remember the visual position of the block.

In case a block defines some of its input parameters as variable length, the number of the actual parameter count is defined through the `vararg` elements, each referencing the parameter `name` and specifying the `count` > 0. For example, a flow definition with a block

```
<block id='Merge1' type='MultiMerge'>
  <vararg name='in' count='5'/>
</block>
```

at compile time will create a MultiMerge block instance with parameters `in1`, `in2`, ..., `in5`. Block bindings then may reference them like any other fixed-parameters.

The `bind` elements connect parameters of various blocks, constants and composite blocks with each other.

```
<bind
  id='string'
  source-block='string'
  source-parameter='string'
  destination-block='string'
  destination-parameter='string'
/>
```

Each binding contains a flow-global unique `id` attribute. The `source-block` attribute references a block or constant identifier within the same enclosing composite block. In case the parent composite block's input needs to be referenced, the attribute should be missing or absent. The `source-parameter` attribute defines the identifier of the block's output parameter or the enclosing composite block's input parameter. In case the `source-block` references a constant, the `source-parameter` may be omitted.

The `constant` elements let the user define constant values to be used as inputs.

```
<constant
  id='string'
  displayname='string'
  documentation='string'
  keywords='string,string,...'
  x='int'
  y='int'
  type='string'
```

```
>
  <!-- Any well formed XML -->
</constant>
```

The id attribute is the flow-global unique identifier of the constant. The displayname attribute can be used when rendering the constant in a GUI environment. The documentation and keywords are the usual information attributes. Attributes x and y is used by the Flow Editor to remember the visual position of the block. Note that the current flow-descriptor implementation doesn't check the type and structure of the constant block values. The type field contains the compile-time type definition of the constant, and has the following format:

```
ns:typename('<' ns:typename ('<' ... '>')? (, ...)* '>')?
```

where the ns:typename references a XSD type. The type definition may contain parametric types introduced by the relation signs (<>), which can contain other types as arguments, and have multiple arguments, even recursively.

2.6.2 Parsing a flow-description XML

A flow-description is managed by the AdvanceCompositeBlock class. The static parseFlow() method parses a flow-description XML and the static serializeFlow() method generates a flow-description XML.

During the parsing process, the flow is verified against some structural requirements. If a requirement is not met, a RuntimeException is thrown.

The most simple error might be the duplicate identifiers used by the main flow-elements (parameters, blocks, constants, bindings, etc.). If one is encountered, a DuplicateIdentifierException is thrown.

In case a composite block's input or output parameter's type definition references an undefined type variable, a MissingTypeVariableException is thrown.

Any other validation is performed only by the compiler as usually more contextual information is required, such as the availability of blocks.

2.6.2.1 Notable classes

The following listings shows the other relevant classes associated with parsing a flow-description and turning it into a tree of Java object.

- AdvanceCompositeBlockParameterDescription: input/output parameter descriptions
- AdvanceBlockReference: the block instance

- `AdvanceConstantBlock`: constants
- `AdvanceBlockBind`: wires between parameters
- `AdvanceBlockVisuals`: general record for storing the x,y coordinates of flow objects.
- `AdvanceTypeVariable`: the definition of the type variables.

2.7 Compiler

The ADVANCE Flow Compiler's main purpose to convert the program from the flow description into runnable, linked network of blocks.

The Compiler is structured in a way that it is independent of the type system and the runtime types of the dataflow network.

The compiler is implemented in the `AdvanceCompiler` class.

Whenever a verification or compilation is performed, the existence and contents of the plugins are re-evaluated (in-process) by the `handlePlugins()` method of the `AdvanceCompiler` class.

2.7.1 API

The compiler's services are accessible through the `AdvanceFlowCompiler` interface.

The interface provides the following methods:

- `verify()` tries to compile a given composite block and returns the detected problems or returns the result of the type inference, thereby providing means to a GUI to show the computed types of the wires between blocks. The results are provided in a `AdvanceCompilationResult` record (2.7.2.7)
- `compile()` compiles and links blocks together and provides a view to these compiled objects through a `AdvanceRealmRuntime` record (2.7.2.6).
- `blocks()` which lists the blocks known by the compiler, which includes the default blocks and the blocks specified inside plugins at the moment (a list of `BlockRegistryEntry`s).

The preparation and teardown of the compiled blocks is arranged via the `AdvanceFlowExecutor` interface through the following methods:

- `run`: initializes the blocks via their `run()` method and sends out startup notifications (i.e., to start constant blocks).
- `done`: terminates the blocks (which may trigger state-saves).

2.7.2 Components and classes

2.7.2.1 AdvanceCompilerSettings

The `AdvanceCompiler` class has a single constructor which takes a `AdvanceCompilerSettings` instance. The class is parametrized by the runtime type (`T`) of the dataflow (`XNElement` in ADVANCE), the type system type (`X`) of the dataflow (`AdvanceType`) and the runtime context type (`C`).

This record class references other classes required during the compilation or provided for the compiled blocks:

- The list of default schema locations (2.3.1.5).
- A map of default blocks and their associated block resolvers (2.7.2.2).
- A plugin manager to resolve external schemas and blocks dynamically (2.7.2.3).
- A map of the actual thread pools by type (`SchedulerPreference` enum).
- A context object provided to all compiled blocks.
- A data resolver that converts from the runtime type values to common Java primitives and structures (2.7.2.4).
- A function-callback to aid the type inference routine extract the type structure and information from the type system type (2.7.2.5).

2.7.2.2 AdvanceBlockResolver

The `AdvanceBlockResolver` interface (and its reference implementation `AdvanceDefaultBlockResolver`) is responsible for locating the block definition and instantiating a block.

The resolver itself is not parsing the `block-registry.xml` files of various blocks but rather keeps a map of block identifiers and their registry entry records, plus contains the Java `ClassLoader` which is used for instantiating the proper class. The class loader (usually a custom `URLClassLoader` is required to load classes from plugin *jar* files which are usually not on the boot classpath of the Flow Engine.

The interface provides the following methods:

- `lookup`: that finds a block definition based on its identifier.
- `create`: instantiates a new block based on its identifier.
- `blocks`: lists the identifiers of the blocks maintained by the resolver.

2.7.2.3 AdvancePluginManager

The `AdvancePluginManager` class is responsible to dynamically locate and interpret external plugins such as blocks and type definitions (XSDs) that may be added at runtime to the Flow Engine (usually no need to restart the application). This is achieved via custom class loader. The detection and preparation of plugins is performed as follows:

- The working directory is listed and files with `*.jar` extensions are checked against the already known plugins (`scanForPlugins()` method).
- If a new plugin is detected or an existing changed, the *jar* file is opened (`processJar()`).
- Internal dependencies are resolved from the `MANIFEST.MF`.

The class implements the `Runnable` interface which allows it to be executed by a scheduled thread pool, which in turn allows the periodic re-checking of the plugin directory.

The class maintains a list of `AdvancePluginDetails` classes that remember each plugin's location, modification date and file size which allow the detection of changes in the plugin. In addition, the `open()` method, returning an `AdvancePlugin` instance opens the plugin *jar* file, parses any block description and collects the type definitions (XSDs). The block description is accessible through the `blockResolver()` method (returning a prepared `AdvanceBlockResolver` instance) and the schema maps through the `schemas()` method.

2.7.2.4 DataResolver

The `DataResolver` is a base interface which allows the development of blocks that do not depend on the runtime type of the compiled dataflow by providing access methods to extract primitive values (boolean, integer, double, string, timestamp, lists, maps, etc.) from the runtime value and create such values from the equivalent Java objects.

For example, in ADVANCE, the runtime type is an `XNElement` in which a simple integer value has the structure:

```
<integer>1</integer>
```

The `AdvanceData` class' `getInt()` method will return the value `1` and the method `create(1)` will create the XML listed.

Working with some generic wrapper types (such as `advance:collection`, `advance:map` and `advance:pair`) requires extra care due to the way the XSDs can't encode generic type information themselves.

For example, in order to statically verify the XML generated of a collection of integers, the collection class `advance:collection` needs to define its child elements with the fixed `item` name and `xs:any` content type (and `xs:anyAttribute`):

```
<collection>
  <item>...</item>
  <item>...</item>
</collection>
```

However, when building such collection XML from a list of integers, the generated XML would look nothing like the XSD specification of the collection type:

```
<collection>
  <integer>1</integer>
  <integer>2</integer>
</collection>
```

The child elements could be renamed to match the XSD, but then the item type information would be lost.

The solution employed by the `AdvanceData` class is to use the `item` as a child node type, but add ADVANCE specific extra attributes that store the original element name and namespace (or a list of them):

```
<collection xmlns:a='http://www.advance-logistics.eu'>
  <item a:original-ns='' a:original-name='integer'>1</item>
  <item a:original-ns='' a:original-name='integer'>1</item>
</collection>
```

where the `original-ns` collects the namespace prefixes in a comma separated list and `original-name` collects the names similarly. The storage of multiple original names is required as in some cases, the end object might be wrapped multiple times.

This wrapping and unwrapping is performed by the `rename()` and `unrename()` methods. To allow comparing such wrapped elements without unwrapping, the `wrappedEquals()` method can be used.

One drawback of the implementation that due to the requirement of immutable runtime objects, copies are created whenever a wrap or unwrap is performed.

2.7.2.5 TypeFunctions

The `TypeFunctions` interface provides means to the type system to support the type inference with various methods. The ADVANCE specific reference implementation class of

AdvanceTypeFunctions does this for the AdvanceType instances of the type system. The interface is parametrized by a type that should extend the Type interface.

The Type interface has only one method: kind() which returns an enumeration TypeKind:

- CONCRETE_TYPE: representing a type, such as integer or string, which is simple and well-defined type.
- PARAMETRIC_TYPE: representing a generic type, such as collection of something.
- VARIABLE_TYPE: which represents an unknown type that could be any of the other kind.

The TypeFunctions requires the following methods to be implemented for a concrete type system:

- intersection: produces a (new) type that contains only the common structure from both input types.
- union: produces a (new) type that contains a combination of structures from both input types.
- compare: returns a TypeRelation enumeration telling if the two input types are equal, one extends the other or have no relation at all.
- setId: assings an unique identifier to type variables so they appear like T[1] in textual output that helps match generic types named similarly across many places. (The type inference considers two type variables the same if they have the same object reference.)
- arguments: returns argument list of a parametric type.
- fresh: creates a new type variable.
- copy: creates a deep copy of the given type (with new type object references).

2.7.2.6 AdvanceRealmRuntime

The AdvanceRealmRuntime record class contains all relevant structures generated by the compiler from a flow description:

- blocks: the list of the compiled and linked blocks.
- inputs: the list of global inputs of the dataflow, mapped by the names of the outermost composite block's input parameters.
- inputTypes: the type of the inputs, mapped by the parameter type.

- `outputs`: the list of global outputs of the dataflow, mapped by the names of the outermost composite block's output parameters.
- `outputTypes` the type of the outputs, mapped by the parameter name.

2.7.2.7 AdvanceCompilationResult

The `AdvanceCompilationResult` class collects the compilation and verification errors of a flow description as well as the inferred types of the wires between blocks and other components.

Several error case may occur during the compilation process, which are reported as the instances of the `AdvanceCompilationError` interface (empty, indicator interface). In addition, many of the error classes implement one or two additional interfaces:

- `HasBinding` indicates that the error has information about the wire causing the problem.
- `HasTypes` indicates that the error has information about the types causing the problem.

The error classes (residing in the `eu.advance.logistics.flow.engine.error` package) are briefly explained in table 29.

Class	Description
`CombinedTypeError`	Reported when two types can't be combined via the union operation (for example, in case of primitive types such as integer and string).
`ConvreteVsParametricTypeError`	Reported when a concrete type is wired into a parametric type (for example, an integer into a collection of string).
`ConstantBlockTypeSyntaxError`	Reported when in the flow description, a constant block's `type` attribute contains invalid contents (2.6.1).
`ConstantOutputError`	Reported when a block's output is bound to a constant.
`DestinationToCompositeInputError`	Reported when a block's output is wired to the input parameter of the enclosing composite block.
`DestinationToCompositeOutputError`	Reported when a block's output is wired to the output parameter of a composite block within the same parent.

Table 29: Compilation errors and their classes

Class	Description
DestinationToOutputError	Reported when two blocks' outputs are wired together.
GeneralCompilationError	Reported when a compilation error deserialized from its XML form references an error class not known by the client.
IncompatibleBaseTypesError	Reported when two different, non-related parametric types are wired together (for example, a collection to a map).
IncompatibleTypesError	Reported when two unrelated types are wired together (for example, an integer to a string).
MissingBlockError	Reported when the flow description references a block which is not known by the compiler (i.e., the flow is aimed at different compiler version or with different plugin settings).
MissingDestinationError	Reported when a wire references a target object identifier not found in the current composite block.
MissingDestinationPortError	Reported when a wire references a parameter (port) not found in the target object.
MissingSourceError	Reported when a wire references a source object identifier not found in the current composite block.
MissingSourcePortError	Reported when a wire references a parameter (port) not found on the source object.
MultiInputBindingError	Reported when multiple outputs are wired to the same input of a block.
MissingVarargsError	Reported when a wire references an item of a variable-length input argument (for example, the wire references in3 but the block has only in1 and in2 arguments).
NonVarargsError	Reported when in the flow description, the block reference sets the number of variable-length arguments on an input parameter that is not declared as variable-length in the block registry.

Table 29: Compilation errors and their classes

Class	Description
SourceToCompositeInputError	Reported when the input of a block is wired to an input of a composite block within the same parent composite block.
SourceToCompositeOutputError	Reported when the input of a block is wired to an output of the enclosing composite block.
SourceToInputBindingError	Reported when an input of a block is wired to an input of another block.
UnsetVarargsError	Reported when one of the variable-lenght parameter is not wired to a block or global input.
UnsetInputError	Reported when a required input parameter is not wired to a block or global input.

Table 29: Compilation errors and their classes

When the compilation errors are serialized, the serialized form contains a `type` attribute which is set up to contain the error class' `getClass().getSimpleName()` value. To deserialize these errors, the `ErrorLookup` class contains a map from these names to a factory that is used for instantiating the proper classes. In case a error type is not known, the default `GeneralCompilationError` is instantiated that contains the raw XML data of the error.

If the compiler encounters an invalid flow description XML (e.g., missing required attributes, non-well-formed XML), those errors are reported as `IOExceptions`.

2.8 XML type system

One of the main results of the ADVANCE project is the invention of a structural type system based on XML Schemas. One of the reasons a simplified type system has been invented is because there were no established methods to compare two XML schemas with each other, let alone, tell if one schema extends the other beyond the explicit schema extension methods.

2.8.1 Representing an XML schema

XML schemas may define wide variety of data types with complex structures. In ADVANCE, the XML type system uses only a subset of the available types and structures of the XML schema and builds mainly on two classes: `XType` and `XCapability`.

2.8.1.1 Limits on the XML schema

The type system does not fully map all features of the XML schema. It lacks the less common features or features which are hard or not possible to be represented in a type system:

- Mixed content is not supported (i.e., when an element contains both text and child elements.)
- Only a subset of the basic XML data types are handled directly, everything else is mapped to string. The `XValueType` enumeration lists the primary supported data types. See 2.8.1.5 about the conversion logic.
- Most constraints, such as uniqueness are ignored.
- The element cardinalities (e.g., `minOccurs` and `maxOccurs`) are interpreted in limited way throug the `XCardinality` enum's values. I.e., limits greater than 1 are interpreted as `unbounded`.
- Attribute and element with the same name are not allowed under an element.

2.8.1.2 Capabilities

XML can be thought of elements containing *attributes*, *text* or other *elements*. The attributes and elements can be interpreted as capabilities of their parent element. Similarly, text content might be yet another capability (if mixed content is allowed) or simply the type of a capability.

Such elements and attributes are represented via the `XCapability` class. The class has the following fileds:

- `name`: The name of attribute/element. The structural comparison (2.8.1.3) relies on this property. It has the `XName` type.
- `cardinality`: defines the occurrence of the element or attribute with the `XCardinality` enum. Attributes are limited to `ZERO`, `ZERO_OR_ONE` and `ONE` values.
- `valueType`: defines if the element or attribute has a simple basic type of `XValueType` supported types. The property is exlusive with the `complexType`.
- `complexType`: indicates the element has its own substructure, which is managed by a `XType` record. The property is exlusive with the `simpleType`.

The `XType` is basically just a list of `XCapability` instances, but it alsow implements the `XComparable` interface, and provides support methods for working with capability lists (`XType`s) required by the type inference algorithm.

- `compareTo`: recursively compares the contents with another `XType`
- `copy`: creates a shallow copy
- `intersection`: creates a new `XType` from the common contents with another `XType`. The resulting `XType`, when compared against the original two, will either yield `EQUAL` or `SUPER`. It may yield effectively an empty `XType` when the two original had nothing in common. In case the original two `XType` had a relation, the one is directly returned which would compare against the other with `SUPER`.
- `union`: creates a new `XType` by combining the contents with another `XType`. The resulting `XType`, when compared against the original two, will yield `EQUAL` or `EXTENDS`. However, it may yield null if the two types can't be properly combined usually due mismatch of simple data types with the same `XName`. In case the original two `XType` had a relation, the one is directly returned which would compare against the other with `EXTENDS`.

In general, both the `XType` and `XCapability` can be the root element of a type structure, ADVANCE chose the former to allow simple representation of an empty type (`Object` in Java terms) that compares against any other, non-empty type as `SUPER`.

Table 30 shows a few examples of types in both XML and in XType (abbreviated as T) and XCapability (abbreviated as C) format.

XML	XType
<object/>	T [C { name: "object", cardinality = ONE, complexType = T[] }]

	T [C {
	name: "a", cardinality = ONE,
	complexType = T [C {
	name: "b", cardinality = ONE,
	simpleType = STRING
	}]
	}]

<a>c	T [C { name: "a", cardinality = ONE, simpleType = STRING }]

<a>
<b/>
<b/>

	T [C {
	name: "a", cardinality = ONE,
	complexType = T [C {
	name: "b",
	cardinality = ZERO_OR_MANY,
	complexType = T []
	}]
	}]

Table 30: Example types in XML and XType form

The XName class represents the textual name of a capability, but features extra properties, such as the semantics and aliases which are used by the comparison method to determine if two capabilities have the same name and meaning. By default, the compareTo() method just compares the string names of the XName instances, but when the extra properties are available, they are considered as well:

- If the semantics (of type XSemantics, which extends XComparable) is present, it may alter the main result of the comparison. In ADVANCE, a default implementation of UriSemantics is available which stores two URI values and may result in EQUAL or NONE relation.

- if the aliases are present (set of strings), then name string of the first is checked against this set, and vice versa. This allows matching, for example, a size named capability with a length capability.

2.8.1.3 XML schema comparison

The XCapability implement the XComparable interface which has a single compareTo() method that lets it compare with another instance of XCapability. The comparison outputs are the TypeRelation enumeration's items.

For the non-recursive case, the comparison method works as follows when started on a XType:

1. Create counters equal, ext and sup that count how many pairs of capabilities were found to be in one of the relations.
2. For each capability of the first type, search the second type's capabilities for an entry with the same name.
3. if found, compare the capabilities (described later). Increment the counters mentioned in step 1 accordingly
4. The final relation is determined by the numers of these counters:
   - let all = equal + ext + sup; let s1 the total number of capabilities of the first type, s2 of the second.
   - if all < s1 and all < s2 then the two types are not not related: NONE
   - if all = equal, i.e., the common capabilities all resolve to equal, then if s1 < s2 then SUPER, if s1 > s2 then EXTENDS, or else EQUAL is the relation.
   - if all = equal + ext and s1 >= s2 then EXTENDS is the relation. I.e., if all common attributes are equal or extends and the first type has at least as many capabilities as the second.
   - if all = equal + sup and s1 <= s2 then SUPER is the relation. I.e., if all common attributes are equal or super and the first type has at most as many capabilities as the second.
   - otherwise, the relation is considered to be NONE.

When two XCapability are compared, the following simpler method is used:

1. Create counters equal, ext and sup that count how many properties were found to be in one of the relations.

2. The `name` properties are compared with their `compareTo` method, and the respective counter is incremented.
3. If one capability has a simple type and the other has complex type, the relation is `NONE`.
4. If both capabilities are of complex type, they are compared via the `compareTo` method, and the respective counter is incremented.
5. The cardinalities of the capabilities are compared, and the matrix in table 31 tells the relation.
6. The final relation is determined by the numers of these counters:
   - let `all = equal + ext + sup`,
   - if `all = equal` then the relation is `EQUAL`,
   - if `all = equal + ext` then the relation is `EXTENDS`,
   - if `all = equal + sup` then the relation is `SUPER`,
   - otherwise, the relation is `NONE`.

Table 31 lists the cardinality relations, using abbreviated regex-like values instead of the long enumeration names of `XCardinality`: `0 = ZERO`, `? = ZERO_OR_ONE`, `* = ZERO_OR_MANY`, `1 = ONE` and `+ = ONE_OR_MANY`.

C1 \C2	0	?	*	1	+
0	EQUAL	SUPER	SUPER	SUPER	SUPER
?	EXTENDS	EQUAL	SUPER	SUPER	SUPER
*	EXTENDS	SUPER	EQUAL	SUPER	SUPER
1	EXTENDS	EXTENDS	EXTENDS	EQUAL	SUPER
+	EXTENDS	EXTENDS	EXTENDS	EXTENDS	EQUAL

Table 31: Relations between the cardinality values

The matrix can be generated via the following simple rules:

- let `n1` the first cardinality enumeration value, `n2` the second
- if `n1 = n2` then the relation is `EQUAL`,
- if `n1.ordinal() < n2.ordinal()` then the relation is `SUPER`,
- otherwise, the relation is `EXTENDS`.

To understand the logic, let's consider the cardinality from a perspective of a function expecting that given number of values (C2). If the function expects 0 values, giving it any number of arguments can be considered as extension. If the function expects 1 value, then giving it one-or-more is considered as extension, but giving it zero-or-more would be unacceptable, hence a restriction.

2.8.1.4 Recursive XML types

The XML schema, and many other kinds type definitions, allow the definition and usage of recursive types, for example, to describe a tree with specific node type. Executing the naive comparison from section 2.8.1.3 would result in an infinite loop / stack overflow.

In many structural type systems, such recursion is managed by keeping track of the so-called *opened types*. If during the comparison, such opened type is found, the comparison is by definition results in an `EQUAL` relation.

Consider the following two XML structures:

```
<xs:complexType name='node'>
  <xs:sequence>
    <xs:element name='node' type='node'
      minoccurs='0' maxoccurs='unbounded'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='node'>
  <xs:sequence>
    <xs:element name='node' type='node'
      minoccurs='0' maxoccurs='unbounded'/>
  </xs:sequence>
  <xs:attribute name='id' type='xs:int' use='required'/>
</xs:complexType>
```

When the outermost `node` is encountered, it is recorded as opened. When the inner `node` is encountered, it is not recursively checked again but automatically reported as `EQUAL`, and in the end, yielding the `SUPER` for the whole comparison.

2.8.1.5 XML schema parsing

The `XSchema` class' parse() method is responsible to turn an XML schema into `XType` structure.

In addition, it offers methods to compare, intersect and union two `XType` instances, where both instances should have zero or one capability. The comparison handles the case where either the

root type is empty or the single capability has an empty complex type.

Since XSD's can reference other XSDs, the external schemas are resolved throug a callback function, receiving a schema location URI. Usually, when local types are resolved, their actual schema files are resolved inside thier respective packages or in the configured schema directories. In general, the `AdvanceDefaultSchemaResolver` manages both standard ADVANCE schema resolutions and external schema lesolutions.

2.8.1.6 ADVANCE default schemas/types

The ADVANCE Flow Engine's type system has several default types available, which are located in the `schemas` project directory, or on the root classpath. Table 32 lists these XSDs.

Type	Description
`object.xsd`	The so-called top type of the type system, but due the way `XType` works, this is just defines an empty `<object/>` element. The comparison methods are set up to handle this and an empty `XType` to be both objects and equal to each other.
`boolean.xsd`	A simple boolean value: `<boolean>true</boolean>`.
`integer.xsd`	A simple integer, arbitrarily large value: `<integer>42</integer>`. This is usually mapped to `BigInteger` within blocks.
`real.xsd`	A simple real number, arbitrary magnitude and precision: `<real>3.14</real>`. This is usually mapped to `BigDecimal` within blocks.
`string.xsd`	A simple textual value: `<string>Hello world!</string>`.
`timestamp.xsd`	A simple, fixed-format date and time value: `<timestamp>2013-08-13T11:14:00.389</timestamp>`
`type.xsd`	Defines a type structure in terms of other XML types, and lets encode the so-called type tokens: e.g., `type<collection<string>>`.
`wrapper.xsd`	Base type that defines the `ns:original-name` and `ns:original-namespace` attributes in types which need to rename contained elements. See section 2.7.2.4 for details.
`pair.xsd`	Represents a simple container with two other wrapped objects.
`option.xsd`	Represents an object that may or may not contain another object, similar to the `Option` class in the Reactive Framework.
`collection.xsd`	Represents a very simple, generic collection of objects with the type. Used by blocks that require list- or array-like inputs and/or outputs.

Table 32: Default ADVANCE types

Type	Description
map.xsd	Technically, represents a collection of pairs of objects: key and value. Note that in ADVANCE, looking up keys is a linear time operation due to the runtime value organization of XNElements.

Table 32: Default ADVANCE types

2.9 Type inference

A graph-based type inference algorithm is used for verifying and evaluating a dataflow program. The inputs are the wires binding various flow objects and their known or generic types. The inference algorithm then tries to figure out the concrete values of the generic types and verify if outputs and inputs wired together are type safe.

2.9.1 Relevant classes

The inference algorithm is implemented in the TypeInference class. The algorithm is independend of the concrete type system used, and only uses a TypeFunctions callback interface to analyze the structures of various types. The only requirement is that the type system instances extend the Type interface.

The inference algorithm stores the type relations in a Relation class which takes the type system type and the wire reference type (which could be anything) as parameters. This allows tracking the type relations in respect of their original wires in the flow description they originate.

The input to the inference algorithm is a sequence of such Relation objects extracted from the flow description. A type relation is a comparison operation between two types (concrete, parametric or variable) and is expressed with the following simple equation:

```
T1 >= T2 (W1)
```

It stores the types as left and right side.

Where T1 is the first type (usually a block's output parameter type), T2 is the second type (usually, a block's input parameter type). The W1 referes to the wire (usually, between two blocks) where the relation originates from.

The result of the inference is stored in a InferenceResult implementation (e.g., in AdvanceCompilationResult. It stores the inferred types of wires and inference errors encountered, such as:

- **Incompatible types**: for example, when an integer is wired into a string.

- **Incompatible base types**: for example, when the base types of two parametric types are not compatible.
- **Argument count mismatch**: for example, when a one-parameter parametric type is wired to a two-parameter parametric type.
- **Combined type**: for example, when two types can't be combined into a single one (usually via union).
- **Concrete vs. parametric type**: for example, an integer is wired to a collection of integer.

2.9.2 Supporting data structures & methods

The graph-based inference algorithm requires several data structures to support the algorithm.

The `upperBound` field, a `MultiMap`, where the key is a type variable (`Type.kind() == VARIABLE_TYPE`), and the multiple values are various kinds of types which represent an upper bound to the type variable. In other terms, if

```
T1 >= T
T2 >= T
advance:string >= T
```

relations are present, the upper bound of the type variable `T` will contain `T1` and `T2`, and if all type variables would be known, then comparing `T` with `T1` would yield `SUPER` (`T1` has at least as many features as `T`). Same would be true for `T2` and `advance:string`.

The `lowerBound` field, a `MultiMap`, does the same thing for lower bounds for type variables:

```
T >= T3
T >= T4
T >= advance:object
```

therefore, the lower bounds for `T` contains `T3` and `T4`. If all types were known, comparing `T` with `T3` would yield `EXTENDS` (`T` has at least as many features as `T3`).

The `reflexives` list collects all unique relations between type variables, i.e., from the previous examples:

```
T1 >= T
T2 >= T
T >= T3
T >= T4
```

(The reflexives list is used by the algorithm to traverse all relations where a particular type variable occurs in case a more concrete knowledge appears about the type variable.)

Finally, the `relations`, a `Deque` contains all remaining type relations that need to be processed. In case a parametric type is encountered, the parameters usually expand into new type relations, which are then added to this queue. The inference algorightm terminates if there were no errors and no more relations remain.

Working with the relations and bounds requires several common methods, found on the `TypeInference` class:

- `containsRelation()` checks if a given list (usually the reflexives) contains any relation where the left and right side have a specific value.
- `mostSpecific()` is a recursive method which tries to determine the 'more concrete' type from two types (`T1` and `T2`)via the following ruleset:
  - If `T1` has a kind of type variable, return `T2`, and vice versa.
  - If `T1` extends `T2`, let `basetype` equal to `T1`, and vice versa.
  - if `T1` and `T2` are parametric types, for each pair of their arguments, execute the `mostSpecific()` method recursively, return a new type with `basetype` and arguments with values returned from the recursive call.
  - if `T2` and `T2` are concrete types, return the `basetype`.
  - if `T1` is a parametric type, return it.
  - otherwise, return `T2`.
- `findParametricBound()` scans a collection of types and returns the first type which is a parametric type.
- `findConcreteType()` for a given type variable, find a concrete type among the upper bounds or lower bounds of the type variable.
- `combineBounds()` given the bounds of a type, adds more bounds to it by combining the concrete types by a function (intersection for the upper bounds, union for the lower bounds).
- `addBound()` adds a single bound to a type, combining the concrete types by a function.

2.9.3 Inference algorithm

The inference algorithm's main method is the `infer()` method, taking and returning an instance of an `InferenceResult` class.

The main loop takes a `Relation` object from the `relations` queue. If the queue becomes empty, the loop terminates, and the original wire types are determined via the `mostSpecific()`

method. Depending on the kinds of the left (L) and right (R) types of the relation, the following cases apply:

- both L and R are type variables: their bounds need to becombined including any reflexive relation they participate in as well.
- L is a type variable and R is a concrete type: L's lower bound must be combined with R, and any relation where L participates.
- L is a type variable and R is a parametric type: L is considered to be a paramteric type, and fresh type variables A1 .. An are paired up with R's arguments, then added to the queue.
- L and R are concrete types: If the comparison does not yields EQUAL or EXTENDS, an error is reported and the inference loop terminates.
- L is a concrete type and R is a type variable: R's upper bound is combined with L, and any relation where R participates as well.
- L is a concrete type and R is a parametric type, or vice versa: is always an error.
- L is a parametric type and R is a type variable: R is considered to be a parametric type and fresh type variables A1 .. An are paired up with R's arguments, then added to the queue.
- L and R are both parametric types: when comparing the base types and getting EQUAL or EXTENDS, the type arguments are paired up and added to the queue.

The bounds logic, in general work on the transitive nature of the type relation. If A >= B and B >= C then A >= C, which means if B receives a new lower bound C, then the lower bounds of A need to be updated as well.

The following subsections explain the cases with examples.

2.9.3.1 L and R are concrete types

The two types are compared through the TypeFunctions callback and the comparison result is SUPER or NONE, an *incompatible types error* is raised through the InferenceResult instance. Otherwise, nothing special happens and the loop continues.

Let T1 := advance:string and T2 := advance:object. If the inference algorithm encountes T1 >= T2 (which compares to EXTENDS), the inference loop continues. However, if T2 >= T1 is encountered (which compares to SUPER) the error is raised and the inference loop quits.

Note that coercions (automatic type conversions) are not handled by the inference algorithm and left for the type system (TypeFunctions) to handle it. In ADVANCE, only the integer to real coercion is performed. Let T3 := advance:int and T4 := advance:real. Comparing T3 and

T4 yields EXTENDS, which is counter intuitive as an int is a restriction over the real numbers, but from function parameter perspective, a function accepting a real number can work with an integer, but not the other way around, hence the EXTENDS relation. Both the intersection and union of T3 and T4 is T4 (real).

2.9.3.2 One is a concrete type, the other is a parametric type

This case yields a *concrete vs parametric type error* through the InferenceResult and the inference loop is terminated. Example: Let T1 := advance:string and T2 := advance:collection<advance:string>; T1 >= T2 is rejected with the error.

2.9.3.3 Both are parametric types

A parametric type consists of a base type and list of type arguments. In case the number of type arguments mismatch, an *argument count error* is raised through the InferenceResult. If the argument count match, the base types are compared, similar to the case of concrete types. (In the current ADVANCE Flow Engine, parametric types only exists in logical form, therefore, a parametric type such as string<string> can be defined and operated upon, but has no real meaning.) I the comparison of the base types yields SUPER or NONE, an *incompatible base types error* is raised through the InferenceResult. Otherwise, new relations are formed from the pairs of the type arguments and added to the relation queue.

Let T1 := collection<string> and T2 := collection<T3>. The two base types are collection which compares to EQUAL. Then, the new relation string >= T3 is added to the main queue.

2.9.3.4 L is concrete- and R is variable-type

This setup is technically the case where the R has a the upper bound L. If R has no concrete upper bounds, the new bound L is set. If R has already an upper bound U, an intersection V := Intersect(U, R) is formed. In case R has a parametric upper bound, the intersection yields object. Usually, the intersection always exists, but only in the form of the object type. The TypeFunctions callback may still yield a null value indicating the non-existence, in which case a *combined type error* is raised in InferenceResult.

In case R participates as the left side in some other type relation (based on the reflexives list), the right sides' upper bounds need to be adjusted as well. For example:

```
T1 >= T2
T1 >= T3
string >= T1
```

In this scenario, `T2` and `T3` have the upper bound `T1`. If the `string >= T1` is encountered, `T1` receives the upper bound `string`. Now since `T1` participates as the left type in the previous two lines, the bounds of both `T2` and `T3` need to be adjusted in a similar way, yielding:

```
string >= T2
string >= T3
string >= T1
```

In case a type variable has an upper bound already, the inference logic plays out as follows:

```
string >= T1
int >= T1
```

When the second relation is reached, `T1` has the upper bound `string`. The new `int` upper bound needs to be intersected with the current `string`, yielding: `object >= T1`.

In the third case, the type `T1` has a parametric upper bound:

```
collection<string> >= T1
string >= T1
```

and the intersection of `collection<string>` and `string` is `object`.

2.9.3.5 L is variable- and R is a concrete-type

This setup is technically the case where `L` has the lower bound of `R`. If `L` has no lower bound, `R` is set as the new lower bound. If `L` has a lower bound `U`, the new lower bound is computed by attempting to union the types: `V := Union(U, V)`. Unlike intersection, the union might not exist, yielding a *combined type error* through `InferenceResult`. (For example, the union of different primitive types might not exist.)

In case `L` participates as the right side in some other type relation (based on the `reflexives` list), the left sides' lower bounds need to be adjusted as well. For example:

```
T2 >= T1
T3 >= T1
T1 >= string
```

In this scenaro, `T2` and `T3` have the lower bound `T1`. When the `T1 >= string` is encountered, all three variable types receive the lower bound `string`.

In case a variable type has a lower bound already, the bounds are combined with union:

```
T1 >= object
T1 >= string
```

where T1's lower bound becomes the union of object and string, which is string.

2.9.3.6 L is variable- and R is a parametric-type

In this case, the L is interpreted as a parametric type by taking the base type of R, the same number of arguments, each with a fresh type variable. Then, new relations with the fresh type variables and the arguments are created and placed in the queue. For example:

```
T1 >= collection<string>
T1 >= collection<T2>
collection<T2> >= collection<string>
T2 >= string
```

In the second line, the T1 gets a lower bound assigned with a fresh T2 variable type, then, the logic with the two parametric types apply (2.9.3.3).

In case a type variable has already a parametric lower bound, the base types and argument count is compared, and *incompatible base types* or *argument count mismatch* errors are raised through the InferenceResult. If the base types compare successfully, the pairs of their arguments are added to the queue.

2.9.3.7 L is parametric- and R is a variable-type

In this case, L is interpreted as a parametric type, similar to 2.9.3.6, with the difference that the upper bounds are adjusted.

```
collection<string> >= T1
collection<T2> >= T1
collection<string> >= collection<T2>
string >= T2
```

2.9.3.8 L and R are both variable types

This is the most complex case of all, since it requries both lower and upper bounds to be updated on all the other relations of L and R as well.

The reflexives list is traversed and checked where L is on the right side and R is on the left side. For example, given a reflexive list:

```
T1 >= L
L >= T2
T3 >= R
R >= T4
```

and the bounds on both variable types:

```
string >= L >= object
object >= R >= object
```

Both lower and upper bounds of both `L` and `R` variable types need to be applied to all places where `L` and `R` appears in a specific position. For example. The lower bounds of `L` need to be applied in those reflexive relations, where `L` is on the right side of the relation: `T1 >= L`. The upper bounds of `L` need to be applied to those reflexive relations, where `L` is on the left side of the relation: `L >= T2`. Due to the transitive nature of `L >= R`, the same logic need to be applied with `R`'s bounds. Therefore, the bounds on `T1` and `T2` become:

```
T1 >= object (from L), object (from R)
string (from L), object (from R) >= T2
```

The logic has to be applied to the `L >= R` bounds as well:

```
string >= L >= object, object (from R)
object, string (from L) >= R >= object
```

After this, all combinations of the lower bound of `L` and upper bound of `R` are added to the queue.

At th end, relation is added to the `reflexives` list.

2.10 Blocks

Blocks are the main programming and computational elements of the ADVANCE Flow Engine. Blocks take inputs and/or outputs, can run asynchronously to each other, and are part of the dataflow network compiled from a flow description. The ADVANCE project has several common functions implemented as blocks, ranging from communication (databases, web services) to data management. In addition due to the logistics environemnt, several specialized blocks are available and usually necessary to perform tasks efficiently. Note, however, that

- Neither the Flow Engine nor the set of blocks were established to be a fully fledged Turing-complete system.
- The granularity of blocks need to be relatively low, i.e., each block performs a rather complex tasks. Simple tasks, such as arithmetics are possible in this setub, but can become

tedious to build in the flow description format. Consider the simple equation `a + 2 * (b - c)`, which would need 3 inputs, 3 arithmetic blocks, a constant block and an output.

Blocks in ADVANCE are derived from the `AdvanceBlock` base class which provides several convenience methods. See section 2.10.2 for further details on block developments.

Table 33 lists the default available blocks in ADVANCE.

Block class	Category	Description
`Alert`	Demo	Displays a dialog that turns a box red if an alert condition is set.
`And`	Projection	Computes the logical AND of the inputs.
`AppendCollection`	Streaming	Appends a new value to the end of a collection.
`AppendMap`	Streaming	Appends a new key-value pair to a map.
`Average`	Aggregation	Computes the average of the values in a numerical collection.
`AverageInteger`	Aggregation	Computes the average of a collection of integers.
`AverageReal`	Aggregation	Computes the average of a collection of reals.
`BayStatus`	Demo	Indicates the filling of bays graphically.
`BufferWithSize`	Streaming	Collects the incoming values into fixed size collections.
`BufferWithTime`	Streaming	Collects incoming values into collections during fixed time windows.
`Button`	Demo	Displays a simple push button GUI.
`Cast`	Projection	Forcibly converts an incoming value to another type.
`CastAll`	Projection	Forcibly converts the elements of a collection into another type.
`Ceil`	Projection	Returns an integer which is strictly greater or equal to the input real.
`CollectionOf`	Projection	Creates a collection from variable number of inputs.
`CollectOptions`	Projection	Collects values with option type into a single regular collection.

Table 33: Default ADVANCE blocks

Block class	Category	Description
ConcatCollection	Streaming	Takes two collections and combines them into one.
ConcatMap	Streaming	Takes two maps and combines them into one.
ConcatString	String	Concatenates two strings.
ConsignmentFilter	Prediction	Filters a stream of consignments based on a date range.
ConsignmentGet	Prediction	Retrieve a collection of consignments from a JDBC data source.
Contains	String	Check if a substring is in a string.
ConvertMapsToObjects	Projection	Converts a collection of maps (key-value) into a collection of objects which contain the key-value data as attributes.
ConvertMapToObject	Projection	Converts a map of key-value pairs into an object with key-value data as attributes.
ConvertOptionMapToObject	Projection	Convert a map of key-value pairs into an object with key-value data as attributes, but if the original option-map is present.
Count	Projection	Counts the elements in a collection.
CreateCollection	Projection	Creates a collection from elements.
CreateOption	Projection	Creates an option-value.
CreateTimedValueGroup	Prediction	Creates a timed-value-group tuple from components.
CreateType	Projection	Extracts the type information from a value.
Crontab	Coordination	Given a schedule, it emits trigger events in a configurable way.
DecodeBase64	String	Decodes a Base64 encoded string.
Dispatch	Projection	Sends out the incoming value into two separate outputs.
DispatchOptions	Projection	Takes a collection and sends out each element as an option, followed by a none-option.
DuringDayConfig	Prediction	Builds the configuration from components for the during-day predictor.

Table 33: Default ADVANCE blocks

Block class	Category	Description
DuringDayModelReader	Prediction	Reads a during-day model XML from a file.
DuringDayModelWriter	Prediction	Writes a during-day model XML into a file.
DuringDayPrediction	Prediction	Calculates the prediction based on configuration and historical consignment information.
DuringDayTraining	Prediction	Trains a model for the during-day prediction.
EmailList	Communication	Returns the list of emails in an inbox.
EmailReceive	Communication	Returns the contents of an email.
EmailReceive	Communication	Returns the contents of multiple emails.
EmailSend	Communication	Sends an email.
EmptyCollection	Projection	Creates an empty collection.
EmptyMap	Projection	Creates an empty map.
EncodeBase64	String	Encodes a string as Base64 string.
EndsWith	String	Checks if a string ends with another string.
FileLogger	File	Appends logging information to a log file.
Filter	Filtering	Applies an XPath to the incoming data and forwards those which matches the XPath.
FilterCollection	Filtering	Filter the elements of a collection with an XPath expression.
FilterMapByKey	Filtering	Filter a map by its keys matching an XPath expression.
FilterMapByValue	Filtering	Filter a map by its values matching an XPath expression.
Floor	Projection	Returns an integer which is strictly less or equal to the real number.
FromCollection	Projection	Converts a collection of values into a single object via an XPath expression.
FromMap	Projection	Converts a map of key-value pairs with XPath expressionts into a new structure.

Table 33: Default ADVANCE blocks

Block class	Category	Description
FTPCreateDir	Communication	Creates a directory on a remote FTP server.
FTPList	Communication	List files on a remote FTP server.
FTPMoveFile	Communication	Move a file on a remote FTP server
FTPMoveFiles	Communication	Move multiple files around on a remote FTP server.
FTPReceive	Communication	Receive the contents of a file on a remote FTP server.
FTPReveiveAll	Communication	Receive the contents of multiple files from a remote FTP server.
FTPReplace	Communication	Replace the contents of a file on a remote FTP server.
FTPSend	Communication	Send a new file to the remote FTP server.
FTPSendAll	Communication	Send multiple files to the remote FTP server.
FTPWatch	Communication	Continuously watch if the contents or files on a remote FTP server directory change.
FullPalletDispenser	Demo	Creates a new full pallet with a random destination.
Gate	Demo	Accumulates messages and sends out them one at a time on the press of a button.
GetItem	Projection	Returns an element from a collection based on an index.
GetKey	Projection	Given a map of key-value pairs, returns a collection of keys where a specific value is mapped.
GetValue	Projection	Returns a value from a key-value map based on a key, or indicates that it was not found.
HalfPalletDispenser	Demo	Creates a new half pallet with random destination.
HubManager	Demo	Collects various pallets, puts them into bays and indicates an alert condition if bays get overloaded.

Table 33: Default ADVANCE blocks

Block class	Category	Description
IndexOf	String	Returns the index of the first occurrence of the substring in a string.
InputBox	Demo	An input box GUI that sends out the string the user entered.
IsEmptyCollection	Streaming	Checks if a collection is empty.
IsEmptyMap	Streaming	Checks if a map is empty.
IsWeekday	Projection	Checks if a timestamp is on a weekday.
JDBCDelete	Database	Issues a *DELETE* statement with a single set of arguments.
JDBCDeleteAll	Database	Issues a *DELETE* statement with multiple sets of arguments.
JDBCInsert	Database	Issue an *INSERT* statement with a single set of arguments.
JDBCInsertAll	Database	Issue an *INSERT* statement with multiple sets of arguments.
JDBCQuery	Database	Issue a *SELECT* statement and return the rows one-by-one.
JDBCQueryAll	Database	Issue a *SELECT* statement and return all rows at once.
JDBCQueryOption	Database	Issue a *SELECT* statement and return each row as an option, followed by a none-option.
JDBCReplace	Database	Issue a *REPLACE* satement with a single set of arguments.
JDBCReplaceAll	Database	Issue a *REPLACE* statement with multiple sets of arguments.
JDBCThrottledBatchQuery	Database	Issue a *SELECT* statement, but return rows in a batch when requested only.
JDBCThrottledQuery	Database	Issue a *SELECT* statement, but return rows when requested only.
JDBCUpdate	Database	Issue an *UPDATE* statement with a single sets of arguments.
JDBCUpdateAll	Database	Issue an *UPDATE* statement with multiple sets of arguments.
JMSQuery	Communication	Perform a JMS query-response operation.

Table 33: Default ADVANCE blocks

Block class	Category	Description
JMSReceive	Communication	Receive messages on a JMS queue.
JMSRespond	Communication	Respond to a JMS query.
JMSSend	Communication	Send a JMS message.
JMSSendAll	Communication	Send multiple JMS messages.
Join	Projection	Combines two objects structurally.
KMeansARXConfig	Prediction	Creates the configuration settings for the K-Means learner/predictor.
KMeansARXLearn	Prediction	Given multiple time series, performs clustering and ARX-based learning.
KMeansARXPredict	Prediction	Runs the prediction based on a single model.
KMeansARXPredictAll	Prediction	Runs predictions based on multiple models.
Lambda	-	Base class that supports running JavaScript as the body of a block.
LastIndexOf	String	Returns the index of the last occurrence of the substring in the string.
Latest	Projection	Remembers the last values of each input and reemits one if a new value arrives in the other.
LocalDirList	File	List the contents of a local directory.
LocalFileLoad	File	Loads the contents of a local file.
LocalFileLoadAll	File	Loads the contents of multiple files.
LocalFileLoadXML	File	Loads the contents of a file as XML.
LocalFileMove	File	Moves a local file.
LocalFileMoveAll	File	Moves multiple local files.
LocalFileOutput	File	Appends content at the end of a file.
LocalFileSave	File	Save data into a file.
LocalFileSaveAll	File	Save multiple data into multiple files.
LocalFileWatch	File	Checks periodically if the contents or files in a local directory change.
Log	Demo	Displays a table of received messages.

Table 33: Default ADVANCE blocks

Block class	Category	Description
MapCollection	Projection	Creates a map from a collection of values by extracting the key via an XPath expression.
MapEntries	Projection	Returns the entries of a map as a collection of pair of key-values.
MapKeys	Projection	Returns the collection of the keys from a map.
MapValues	Projection	Returns the collection of values from a map.
Marshall	Projection	Converts an object into an XML representation and returns it as string.
Max	Projection	Returns the largest element of a collection.
MaxAll	Projection	Returns all maximum elements of a collection.
MaxInteger	Projection	Returns the largest integer value of a collection.
MaxIntegerAll	Projection	Returns all maximum integer values of a collection.
MaxReal	Projection	Returns the largest real value of a collection.
MaxRealAll	Projection	Retursn all maximum real values of a collection.
Mean	Projection	Computes the mean of the values in a collection.
Merge	Streaming	Merges two streams of values into one.
Min	Projection	Returns the smallest element of a collection.
MinAll	Projection	Returns all minimum elements of a collection.
MinInteger	Projection	Returns the smallest integer value of a collection.
MinIntegerAll	Projection	Returns all minimum integer values of a collection.

Table 33: Default ADVANCE blocks

Block class	Category	Description
MinReal	Projection	Returns the smallest real value of a collection.
MinRealAll	Projection	Retursn all minimum real values of a collection.
MultiMerge	Streaming	Merges multiple data streams into one.
Not	Projection	Negates a logical value.
OptionValue	Projection	Extracts a contained value from an option.
Or	Projection	Computes the logical OR of the inputs.
ParseInt	Projection	Converts a string into an integer.
ParseReal	Projection	Converts a string into a real.
Project	Projection	Extracts a substructure with an XPath expression.
RegexpMatch	Projection	Runs a regular expression on a string and returns the matched groups.
RegexpMatches	Projection	Check if a string matches a regular expression.
RelayIf	Streaming	Relays a value in case a condition is true.
RelayIfTriggered	Streaming	Relays a value in case another value arrives as well.
Remap	Projection	Remaps the keys of a map with an XPath expression.
RemoveEntry	Projection	Removes a key-value pair from a map.
RemoveIndex	Projection	Removes an element at a specified index from the collection.
RemoveIndexRange	Projection	Removes elements from an index range from a collection.
RemoveItem	Projection	Removes an element from a collection.
RemoveKey	Projection	Removes a key-value pair based on a key.
RemoveValue	Projection	Removes all key-value pairs based on a value.
Reverse	Projection	Reverses a collection.
Round	Projection	Rounds a real value to the nearest integer based on the half-rule.

Table 33: Default ADVANCE blocks

Block class	Category	Description
RunningExtremes	Projection	Keeps track of the minimum and maximum values received.
RunningSum	Projection	Returns a running sum of the input sequence.
SelectAttribute	Projection	Extracts the value of an attribute with the given name from the input structure.
SelectChild	Projection	Extracts a child-structure with the given name from the input structure.
SelectChildren	Projection	Extracts all child-structures with the given name from the input structure.
SimpleRunningStatistics	Projection	Keeps track of running statistics of the input sequence.
Singleton	Projection	Creates a collection with a single element.
SingletonMap	Projection	Creates a map with a single key-value pair.
SMSSend	Communication	Send an SMS.
SOAPInvoke	Communication	Invoke a method through a SOAP call.
SOAPReceive	Communication	Receive a SOAP request.
SOAPRespond	Communication	Respond to an incoming SOAP request.
SOAPSend	Communication	Send a SOAP message.
SplitEvent	Projection	Split a collection into a given number of sub-collections evenly.
SplitSize	Projection	Split a collection into given sized sub-collections.
StartsWith	Projection	Checks if a string starts with another string.
STDDeviation	Projection	Computes the standard deviation of a value collection.
STDDeviationInteger	Projection	Computes the standard deviation of an integer collection.
STDDeviationReal	Projection	Computes the standard deviation of a real.
Substring	Projection	Extracts a sub-string from a string.

Table 33: Default ADVANCE blocks

Block class	Category	Description
Sum	Projection	Computes the sum of the values of a number collection.
SumInteger	Projection	Computes the sum of the integers in a collection.
SumReal	Projection	Computes the sum of reals in a collection.
Timer	Demo	Periodically emits the last value it received on its input.
ToBoolean	Projection	Convert a string into a boolean value.
ToCollection	Projection	Convert the elements of a structure into a collection.
ToInteger	Projection	Convert a string into an integer value.
ToLowercase	Projection	Convert a string into lowercase.
ToMap	Projection	Convert a structure into a map via key and value XPath expressions.
ToReal	Projection	Convert a string into a real value.
ToString	Projection	Converts a simple type value into string.
ToTimestamp	Projection	Convert a string into a timestamp.
ToUppercase	Projection	Convert a string to uppercase.
Transform	Projection	Applies an XSLT transformation on a structure.
TruckLoader	Demo	Loads up a truck with pallets.
Unmarshall	Projection	Converts a XML-string representation back to an object.
Unwrap	Projection	Takes elements from a collection and dispatches them one-by-one.
WebGET	Communication	Issues a HTTP *GET* to a web data source.
WebPOST	Communication	Issues a HTTP *POST* to a web data source.
WebReceive	Communication	Awaits a HTTP message.
WebRespond	Communication	Responds to a received HTTP message.
Where	Streaming	Evaluates a JavaScript condition for each incoming value.

Table 33: Default ADVANCE blocks

Block class	Category	Description
`WithDefault`	Demo	Block for testing the default input.
`Wrap`	Projection	Wraps the inputs into a collection.
`WriteLine`	Demo	Writes a string to *STDOUT*.
`Xor`	Projection	Computes the logical XOR of the input parameters.

Table 33: Default ADVANCE blocks

A basic, generic typed block class looks like this:

```
@Block(id="DuplicateExample", category="demo", scheduler="IO",
description="Example block that duplicates ", parameters={"T"})
public class DuplicateExample extends AdvanceBlock {
  @Input("?T")
  protected static final String VALUE = "value";
  @Input("advance:integer")
  protected static final String COUNT = "count";
  @Output("advance:collection<?T>")
  protected static final String OUT = "out";
  @Override
  protected void invoke() {
    XNElement value = get(VALUE);
    int count = getInt(COUNT);
    List<XNElement> result = new ArrayList<>();
    for (int i = 0; i < count; i++) {
      result.add(value);
    }
    XNElement out = resolver.create(result);
    dispatch(OUT, out);
  }
}
```

The methods, annotations and structures used by the block are described in the following sections.

Note that naming of blocks (`id`) should be unique since the Flow Engine does not support any kind of overload resolution, i.e., blocks with the same name but different parameters. The reason for this is manyfold:

- The type inference algorithm is currently not suited for the case.
- Adding overload resolution to type inference usually makes it slower, and adds the burden for the user to handle ambiguous cases.
- When deploying blocks in the Flow Editor, the block's identifier has to be known so the proper number of parameters are placed and can be wired.

2.10.1 Block annotations

Blocks descriptions may be created automatically from the source code using ADVANCE specific annotations. The Java build process must be configured to process annotations an generate the output XML file. In Eclipse, open the properties sheet for the project and then select the **Java**

Compiler section on the left tree (see Figure 2). Open this tree item in order to reveal the **Annotation Processing** and **Factory Path** items. In the **Annotation Processing** sheet you need to first enable the project specific settings checkbox (if not already enabled) and then enable both the annotation processing and processing in editor options. Then you need to set the **Generated source directory** field to the **schemas** text.

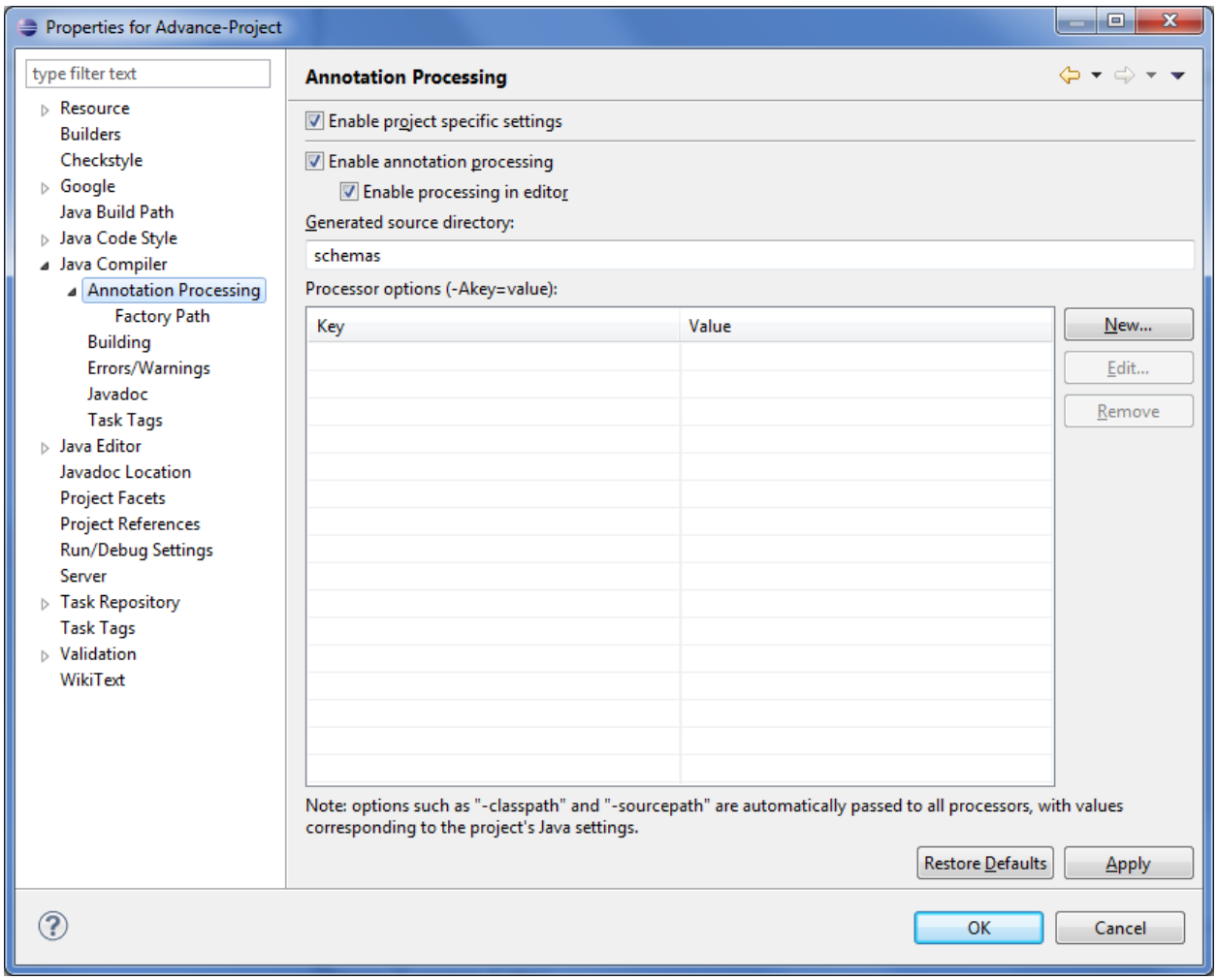

Figure 2: Annotation processing properties

In the **Factory Path** sheet you need to enable the project specific settings (if not already checked) and then add the **advance_apt.jar** plug-in using the **Add JARs** button and selecting the file from the project **lib** directory (see Figure 3).

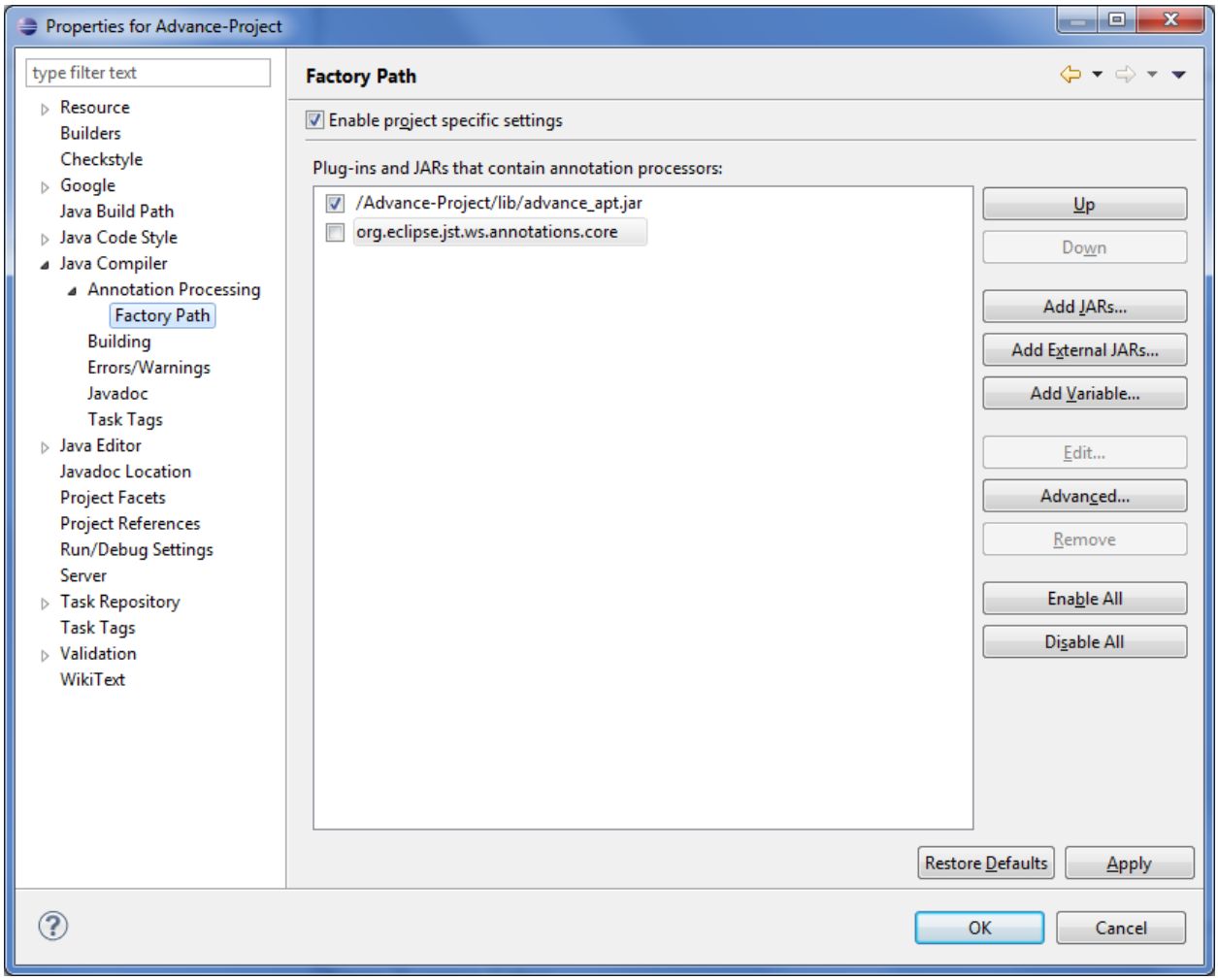

Figure 3: Annontation processors factory path

By default, the block registry in `schemas/block-registry.xml` gets updated partially if a block is saved in the IDE (because the IDE only calls the annotation processor for that single file, in which case recreating the whole block registry would delete the other entries). If blocks get deleted, that leaves their registry entries in place, therefore, to remove all unused entries, or just regenerated the whole registry, open the `block-registry.xml` and delete everything, then run a full project rebuild (Clean in Eclipse). Do not delete the `block-registry.xml` as it can cause problems if the file is versioned.

Developers may annotate their block classes and and block fields. These annotations are then translated by the IDE (Integrated Developer Environment) into the block-registry resource.

The annotation facility consists of three annotations, described in the following subsections. The annotations are located in the `eu.advance.logistics.annotations` package.

2.10.1.1 Block

Let's the user define the basic block properties, such as the unique identifier, the scheduler to be used, keywords, category and short description to be displayed in the Flow Editor. In addition, the developers need to specify the type variables of the inputs and outputs up front. Example:

```
@Block(id = "Average",
  category = "aggregation",
  scheduler = "IO",
  description = "Compute the average of the integer or
     real values within the collection",
  keywords = "kw1, kw2",
  documentation = "http://www.advance-logistics.eu/
    doc/fe/blocks/Average",
  parameters = { "T" }
)
public class Average extends AdvanceBlock { }
```

The `id` is the unique identifier of the block, the flow description's block `type` will reference it. The `category` is used by the Flow Editor to group blocks together. The `categories.xml` lists the editors default categories (which have custom icons). Categories not found in this file are added with default icons. The default categories are listed in table 34.

Category	Annotation value	Icon
Data transformation	`data-transformations`	
Data normalizing	`data-normalizing`	
Data generation	`data-generation`	
Arithmetic	`arithmetic`	
Classification	`classification`	
Statistics	`statistics`	
Projection	`projection`	
Aggregation	`aggregation`	
Optimisation	`optimisation`	
Simulation	`simulation`	

Table 34: Default categories of the Flow Editor

Category	Annotation value	Icon
Self Learning	`self-learning`	
User Interface	`user-interface`	
Combinatorial optimisation	`combinatorial-optimisation`	
Travel cost matrix / interface	`travel-cost`	
Uncategorized	any	

Table 34: Default categories of the Flow Editor

The `scheduler` attribute determines the execution of the block and may take the following values (the enum names of the `SchedulerPreference` but due package cross-reference issues, they are defined as plain strings):

- `CPU`: the block runs on a thread pool which utilizes all available CPUs.
- `IO`: the default, and larger thread pool suitable for long running and mostly blocking operations.
- `SEQUENTIAL`: the block runs in a FIFO-like thread pool which ensures that blocks execute sequentially.
- `NOW`: the block runs in the same thread pool of the last block to avoid unnecessary context switches.

The `description` field contains a short sentence about the purpose of the block, which is shown by the Flow Editor as a tooltip. The `keywords` allow the developer to list free-text, comma separated keywords that may help find the block. The `documentation` contains an URI referencing a documentation resource.

The `parameters`, if present, list the type arguments required by the inputs and outputs of the block with the following format:

```
name ('+'?upperbound | '-'lowerbound)? (, ...)*
```

where the optional `upperbound` and `lowerbound` can be type definitions, potentially referencing other type variables:

```
typename ('<' typeargument (, typeargument)* '>')?
```

where the typename is any known ADVANCE type, type variable (prefixed by ? to distinguish between simple types and type variable names), or external data type and the typeargument is a contains the whole definition again recursively.

Tabel 35 lists a few examples of type definition:

Example	Explanation
T	A simple generic type variable without bounds.
T -advance:string	A generic type with string lower bound.
T +advance:pallet	A generic type with a pallet as an upper bound type.
T -advance:map, -advance:collection	A generic type with combined lower bounds of map and collection.
T -?U	A generic type which has another generic type as lower bound.
T -advance:collection<advance:string>	A generic type which should extend the base type *collection of string*.
T -advance:map<advance:string, ?U>	A generic type extending the base type of *map* with string key and another generic type as value.

Table 35: Type declaration examples for the Block annotation.

2.10.1.2 Input

Annotates a constant string field denoting an input parameter and describes its properties, such as the type.

When the block registry is built, the id of the input parameter is derived from the annotated constant Java String typed field's value.

In addition, the developers may define a default value for the input (which is translated to a constant by the compiler in case no block binding is defined by the flow-descriptor), and mark it as essential input (the users of the Flow Editors must wire something in). The parameter may represent a variable number of input arguments; when the user in the Flow Editor places such blocks, he or she must specify the number of inputs to create for that particular type. For example, the MultiMerge block let's the user specify the number of input arguments, some might have 2, some might have 20 inputs. Placing such blocks in the flow will allow the user to wire in as many inputs as available. Variable inputs use the input name plus a numerical postfix

starting from one. Type variables can be referenced by using a question mark (?) before the type variable name to ensure it is not confused with a regular XML schema type. Example:

```
@Input(value = "?T",
  variable = true,
  required = false,
  defaultConstant="<raw>xml</raw>"
)
protected static final String IN = "in";
```

The default annotation parameter value contains the type of the parameter. It can reference a type variable definition (2.10.1.1) or specify a type directly:

```
('?'typevariable | typename ('<'typeargument (, typeargument)* '>')?
```

Table 36 lists a few examples of the type definitions.

Example	Explanation
?T	Reference a block type variable.
advance:string	A simple type.
advance:collection<?T>	A parametric type where the type argument is a type variable.
advance:map<?T, ?U>	A parametric type with both type arguments as type variables.
advance:collection<advance:collection<?T»	A multi-level parametric type.

Table 36: Type declaration examples for the Input annotation

The variable annotation parameter indicates that the given block input parameter is a variable length (2.6.1) parameter, i.e., depending on the instantiation, the block can take 1, 2, ..., n parameters. The required annotation parameter indicates that the block input parameter must be wired in. The defaultConstaint contains a raw XML to be used as the default constant input for the given block input when it is not wired to something else. Developers should take caution as (currently) there is no type verification of this raw XML at development time or runtime.

2.10.1.3 Output

Annotates a constant string field denoting an output parameter. It describes the output type and the optional notion of variable number of outputs.

When the block registry is built, the `id` of the output parameter is derived from the annotated constant Java `String` typed field's value.

Example:

```
@Output(
  value = "?T",
  variable = false
)
private static final String OUT = "out";
```

The annotation default `value` parameter defines the output type, similar to the `Input` annotation (see section 2.10.1.3 and table 36). The `variable` annotation parameter indicates that the given block output parameter is a variable length (2.6.1) parameter, i.e., the block can provide 1, 2, . . . , n parameters.

2.10.1.4 Block annotations subproject

The ADVANCE project contains an annotation subproject which contains the code that produces the `block-registry.xml` from the annotations (2.10.1) in the main Flow Engine project. The bundled `advance_apt.jar` package contains the annotations and the processor logic.

The main logic is in the `BlockDescriptionGenerator` which extends the Java standard annotation processing framework.

The main entry point is the `process()` method. It parses the block registry as raw text file and builds a map with the block identifiers and the raw block body XML. The text processing is looking for the `"<block-description"` substring, which should not occur as regular values inside block declarations (i.e., as a default value constant).

Then the target classes are analyzed and the annotations are turned into the XML descriptions with the format specified in section 2.3.2.

The last step then saves the entire map of the registry (with the modified and unmodified block definitions) into the `schemas/block-registry.xml` file.

In case a processing error occurs, the standard `javax.annotation.processing.Messager` is notified with the problem description, which is visible in Eclipse throught the *Window* / *Show view* / *Error log* view.

2.10.2 Developing new blocks

The `AdvanceBlock` abstract class contains a skeleton for concrete block implementations. All internal ADVANCE blocks are derived from this class and they all apply some form of customization over this class.

The primary extension point is the `invoke()` method. Implementations override this method to add their functionality. The method doesn't need to be synchronous, but in order to create outputs synchronously or asynchronously, the submission of output values is performed by the `dispatchXYZ()` methods.

Accessing input parameters (constant or reactive) can be done via `getXYZ()` methods:

```
String value = getString(INPUT_PARAMETER)
XNElement value = get(XML_INPUT_PARAMETER)
```

The default behavior of waiting for all inputs to become available is ensured in the `runReactiveBlock()` method. Implementations may override this method if they want a different triggering behavior, e.g., they only want to wait for some ports only, or do not want to wait at all. One example for this behavior is the MERGE block, which joins two streams of values into one single stream and relays values as soon as they arrive at any of its input ports.

If the `runReactiveBlock()` is overridden, the developers usually need to override the `runConstantBlock()` method, which handles the special case when all input ports are bound to constants. As mentioned before, these blocks are only executed once by the runtime, and some blocks may be implemented a bit differently for this case. The MERGE block, for example, needs to gather all the constant values and send them out one after another in this 'jump-start' scenario and never run again.

Convenience methods helping in the block's internal execution are detailed in section 2.10.2.6.

Each `AdvanceBlock` contains a context object, `AdvanceRuntimeContext` which allows the implementation to access common system objects through the settings field:

- The available schedulers (notably the CPU and IO schedulers).
- The source and position of the block (e.g., in which realm, in which composite block it is).
- Resolver for basic datatypes (e.g., convert to and from XML representation).
- The datastore interface.
- The connection pool manager (e.g., database connection pools, etc.).

Each of these context elements can be accessed though the `settings` field of the `Block` class, which field is a `BlockSettings` record. In ADVANCE the context inside the settings is of

type AdvanceRuntimeContext. Components of the settings and context are described in the following subsections.

2.10.2.1 Schedulers

The BlockSettings features a schedulers map, keyed by the SchedulerPreference. Tasks in form of Runnables can be submitted to it through the Scheduler interface.

Each task submitted to a scheduler will result in a Closeable instance. Blocks should remember such instances in order to stop a future task or cancel a periodic task.

The blocks themselves run on a scheduler, which can be accessed through the scheduler() method on the Block.

Note that when the Flow Engine exits, it shuts down the schedulers, interrupting any running operations. Block implementations should take care implementing a proper done() method which, if necessary, stops any spawn tasks and saves any state required.

2.10.2.2 Flow description relation

The BlockSettings contains references to the flow description which the block was compiled from in form of the parent composite block (AdvanceCompositeBlock) and the block reference inside the composite block (AdvanceBlockReference). The blocks own block-registry definition is also available (BlockRegistryEntry). The current realm the block is running in is stored in the realm field.

2.10.2.3 Data resolver

The runtime data types are fixed for the entire Flow Engine, and for convenience, a DataResolver is provided to each block, which lets develop block logic independent of the runtime data types. The data resolver is in the resolver field, but convenience access is provided through the Block class. See section 2.7.2.4 for further details.

2.10.2.4 DataStore

The access to the AdvanceDataStore is provided through the context field of the BlockSettings, which context is an instance of AdvanceRuntimeContext. Blocks can access data source details, load and save themselves as necessary. See section 2.4 for details on the DataStore API.

2.10.2.5 Connection pools

Many external connections need to be pooled due to performance reasons or capacity limits. The `BlockSettings.context.pools` contains the `AdvancePools` instance, which manages the handing out of connection objects to various resources (such as JDBC connections, JMS connections, etc.).

A connection `Pool` object for a specific resource identifier can be obtained via the `get()` method. The method takes a class and an identifier. The class is used to determine the resource type and the identifier looks up the connection information in the DataStore. Many classes can be used to retrieve a pool, which are mapped in the `AdvancePoolCreator` instance, initialized during the engine configuration. Table 37 lists the classes and the pool type returned by the `get()` method. In general, pools can be requested based on the connection object's class and the ADVANCE datastore record class (see table 6, marked with * in table 37).

Classes	Pool	Pool manager

`JDBCConnection`
`java.sql.Connection`
`AdvanceJDBCDataSource`*
	`BoundedPool` of `JDBCConnection`	`JDBCPoolManager`

`FTPConnection` `ADvanceFTPDataSource`*	`UnlimitedPool` of `FTPConnection`	`FTPPoolManager`
`JMSConnection` `AdvanceJMSEndpoint`*	`BoundedPool` of `JMSConnection`	`JMSPoolManager`

`LocalConnection`
`java.io.File`
`AdvancLocalFileDataSource`*
	`UnlimitedPool` of `LocalConnection`	`LocalPoolManager`

`SOAPConnection` `AdvanceSOAPEndpoint`*	`UnlimitedPool` of `SOAPConnection`	`SOAPPoolManager`
`WebConnection` `AdvanceWebDataSource`	`UnlimitedPool` of `WebConnection`	`WebPoolManager`
`EmailConnection` `AdvanceEmailBox`	`UnlimitedPool` of `EmailConnection`	`EmailPoolManager`

Table 37: Connection pool supported classes

2.10.2.6 AdvanceBlock methods

The `Block` and `AdvanceBlock` classes provide several convenient methods that help access to parameters, configuration and context. Table 38 lists the most relevant methods.

Method	Description
`init`	Initializes the constant and reactive input/output ports.
`getReactivePorts`	Returns those input ports, which don't have a constant input value.
`getConstantPorts`	Reutns those ports which are wired to a constant value.
`run`	Schedules the execution of the body function; can be overridden to replace the default behavior of reacting to inputs.
`runReactiveBlock`	Sets up a collector function that reacts only when all inputs have arrived on the reactive ports.
`runConstantBlock`	Schedules the execution of the block body in case all inputs are constants.
`invokeBody`	Combines the constant port values with the values of the reactive ports into the `params` map and calls the `invoke()` body function.
`log`	Logs an exception both on the logging framework and the block diagnostic port.
`dispatch`	Send out values to output ports.
`invoke`	The body function, needs to be overridden by concrete blocks.
`done`	Shuts down the block and disconnects all input ports.
`description`	Returns the block's registry definition.
`getDiagnosticPort`	Returns an observable that notifies observers of main block events.
`getOutput`	Returns a named output port.
`getInput`	Returns a named input port.
`saveState`	Called by the Flow Engine to ask the block to produce a state XML.
`restoreState`	Called by the Flow Engine to ask the block to restore its state (after engine restart). If the flow program changes in a realm, this is not called.

Table 38: Block convenient methods

Method	Description
scheduler	Returns the Scheduler defined by the block's definition. Scheduling and synchronously waiting for a task to complete is not recommended due to threadpool-deadlocks.
id	The block's unique identifier as it was declared in the flow description.
parent	Returns the parent composite block from the frow description.
addCloseable	Adds a closeable instance to the internal functionClose list, that are automatically closed if the block is terminated.
inputs	Returns all input ports.
outputs	Returns all outputs.
get getBoolean getInt getDouble getString getTimestamp	Returns (interprets) a value from the named input port, (handles constant and reactive values automatically).
getAll	Returns the values of a varargs parameter as a single list.
set	Several overloads produce a value and send it to a named output.
register	Takes a port and observer, and registers on the port and observers it on the block's preferred scheduler.
resolver	Returns the data resolver instance.
observeInput	Registers an individual observer action on an input port.
varargs	Returns all values of a varargs parameter.
getPool	Returns a pool for a given class and connection configuration id.

Table 38: Block convenient methods

2.11 Plugins

In order to add blocks dynamically to a (running) Flow Engine, a plugin system was devised. JAR files need to be placed into the Engine's plugin directory (usually under the plugins directory in Engine's working directory, see the engine configuration in section 2.3.1.7).

The following listing shows the files in an example plugin JAR:

```
block-registry.xml
type1.xsd
type2.xsd
META-INF/MANIFEST.MF
com/example/blocks/exampleblock1.class
com/example/blocks/exampleblock2.class
```

A plugin is recognized by having a `block-registry.xml` in its root directory (section 2.3.2). XML Schema files need to be placed under the same root directory. In case the file is malformed or any of its blocks can't be resolved, the plugin is ignored.

The plugin may contain the regular `META-INF/MANIFEST.MF` file, which gets parsed, and any referenced library references loaded as well from the `Class-Path` parameter. Files are resolved against the plugin directory, e.g., the entry

```
Class-Path:  lib/lib1.jar
```

is resolved as `workdir/plugins/lib/lib1.jar`.

The `AdvancePluginManager` loads libraries into memory, therefore, the underlying plugin files can be removed any time. The manager will unload the plugin on the next refresh cycle.

2.12 Running the Flow Engine

The Flow Engine can be run in three different modes:

- standalone server,
- embedded full mode and
- embedded verifier mode.

Each of these modes are detailed in the following subsections.

2.12.1 Standalone server

In *standalone server* mode, the Flow Engine sets up a HTTPS server which can be accessed by using the XML-based API described in section 2.5.1.1.

The simplest way to start the server is to invoke the `AdvanceFlowEngine.main()` which, by default, works from the user's home directory in `.advance-flow-editor-ws` subdirectory. The `main()` method may take an optional parameter, specifying the configuration file. An additional parameter may tell the Engine to use a specific working directory:

```
java -cp advance-flow-engine.jar
  eu.advance.logistics.flow.engine.AdvanceFlowEngine
  /conf/flow_engine_config.xml /var/advance/workdir
```

The format of the configuration file is described in section 2.3.1. Here is a simple example configuration file:

```
<?xml version='1.0' encoding='UTF-8'?>
<flow-engine-config>
  <listener cert-auth-port='8443' basic-auth-port='8444'
    server-keystore='DEFAULT'
    server-keyalias='advance-server' client-keystore='DEFAULT'
    server-password='YWR2YW5jZQ==' />
  <datastore driver='LOCAL' url='datastore.xml' />
  <keystore name='DEFAULT' file='keystore' password='YWR2YW5jZQ==' />
  <scheduler type='CPU' concurrency='ALL_CORES' priority='NORMAL' />
  <scheduler type='IO' concurrency='16' priority='NORMAL' />
  <scheduler type='SEQUENTIAL' concurrency='1' priority='NORMAL' />
</flow-engine-config>
```

The Flow Engine can be more directly invoked via the following code snipplet:

```
AdvanceFlowEngine afe = new AdvanceFlowEngine(configFile, workDir);
afe.run();
```

The engine may be terminated via the `shutdown()` method. Accessing the Engine API and DataStore API is possible through the `control()` and `datastore()` methods respectively.

2.12.2 Embedded full mode

In *embedded full mode*, no HTTP server is started and the engine's functions are accessed through direct API calls. Such mode can be initialized with the following sequence of statements:

1. Create and initialize the configuration:

```
XNElement configXML = XNElement.parseXML(configFile);

AdvanceEngineConfig config = new AdvanceEngineConfig();
config.initialize(configXML, workDir);
```

2. Create and initialize the compiler:

```
AdvanceCompilerSettings csettings =
  config.createCompilerSettings();
AdvanceCompiler compiler = new AdvanceCompiler(csettings);
```

3. Initialize the datastore and create the flow engine:

```
AdvanceDataStore ds = config.datastore();
AdvanceEngineControl engine = new LocalEngineControl(ds,
  config.schemas, compiler, compiler, workDir);
```

4. Create a view into the Engine which checks for user rights:

```
AdvanceEngineControl cengine = new CheckedEngineControl(engine,
  userName);
```

Remark: type parameters were omitted for clarity.

The Engine instance may be shut down through the `engine.shutdown()` method, but since the datastore is managed by the `AdvanceEngineConfig` instance, that should be shutdown as well with its `close()` method.

2.12.3 Embedded verifier mode

In the *embedded verifier mode*, only the compiler is initialized and only its verification capabilites are available. This mode is used by the Flow Editor to verify the user-built flow in a real-time manner. This mode can be started with the following steps:

1. Create and initialize the configuration:

```
XNElement configXML = XNElement.parseXML(configFile);

AdvanceEngineConfig config = new AdvanceEngineConfig();
config.initialize(configXML, workDir);
```

2. Create and initialize the compiler:

```
AdvanceCompilerSettings csettings = config
  .createCompilerSettings();
AdvanceCompiler compiler = new AdvanceCompiler(csettings);
```

3. Call the compiler to verify a flow:

```
compiler.verify(flow);
```

The compiler does not need to be closed or terminated in any way.

2.12.4 BasicLocalEngine

The `BasicLocalEngine` utility class helps in instantiating the embedded mode Flow Engines in a convenient way through the following methods:

- `create()` will create a local engine based on a simple configuration listed in 2.12.1 and using the provided working directory.
- `defaultConfigFile` returns the default config as string.
- `defaultConfig` initializes a configuration with defaults.
- `createCompiler()` crates a basic compiler with the default ADVANCE blocks and schemas.

The engine created via the `create()` method (a `CheckedEngineControl` instance) has an *admin* user with the password *admin*.

3 ADVANCE Flow Editor

3.1 Overview

The Flow Modeller is developed using the NetBeans Rich Client Platform (RCP) as a modular application. The first and most important consequence of this is that the NetBeans IDE is required to develop extension or change the source code of the Flow Modeller. Currently the Flow Modeller is based, and thus requires, version 7.3. In this chapter we will give a general overview of how the application is organized and what are its extension points (i.e. where external developers can add their functionalities). For more information of how the NetBeans RCP works, how you can develop modules you can refer to the official documentation at www.netbeans.org. Being based on the NetBeans RCP it is very easy to add additional modules in order to increase the feature set of the application.

3.1.1 NetBeans Rich Client Platform

A NetBeans RCP application is a composed of modules discovered and loaded at runtime. They can install various bits of functionality, such as components, menu items, or services; or they can run code during startup to initialize programmatically; or they can take advantage of declarative registration mechanisms that various parts of the platform to register services and initialize them on demand. The NetBeans Module System uses the declared dependencies of the installed components to set up the parent classloaders for each module's own classloader, determining what JARs will be searched when a module tries to load a class.

When you develop on the NetBeans RCP, you get a rich set of base features, extensive APIs with which you can create your own features, and a powerful set of tools to help you in development. The following are some of the most important standard modules used in the Flow Modeller application:

- Window System: greatly simplifies the manipulation of multiple components within a single frame.
- Actions system: makes it easy to declaratively install and uninstall menu items, toolbar items, keyboard shortcuts.
- Auto Update mechanism: provides a way to dynamically update the application.
- Visual Library: provides a set of reusable widgets and a graph model composed of nodes, edges and pins.
- Tree view, Nodes and property sheets: various modules that provides a model to display a hierarchy of objects as nodes in tree, with optionally some associated properties.

3.1.2 Application Structure

The Flow Modeller is currently composed of an application suite (i.e. a group of modules plus additional customizations) and three modules, two of which constitutes the essential functionalities of the application and the other one is an extension that provides integration with the Flow Engine Control Centre. The main project is named **Advance Flow Editor** and you can open it in the NetBeans IDE to develop and debug your code. Be sure to check the **open dependencies** check box in order to open the three modules, as they are declared as sub projects of the application suite. You should have all the projects opened as show in Figure 4. The Flow Modeller is configured as a standalone application and depends only on the **platform** module from the standard installation (plus the three custom modules).

3.1.3 Branding

The customization of the appearance and texts of the application is done using the **Branding...** menu item in the project context menu. Currently we provide a custom icon in all the required sizes (16x16, 32x32 and 48x48). These icons are used in the application window title and in the operating system taskbar/dock. A custom splash screen with the **ADVANCE** project logo is also included. We have used the default values for the Window System enabled features. In the **resource bundle** it is possible to customize texts that are defined in the standard platform library, effectively overriding their value to adjust to the application context. For example we changed the exit message from **Exit IDE**, that is the default message for the NetBeans Platform, to just **Exit**, that is more appropriate for the Flow Modeller application.

3.1.4 Declarative Registration

The base application framework already provides most menus and toolbar buttons for many of the common operations, such as copy, cut, paste and exit. Specific actions con be registered using the declarative registration mechanism provided by the NetBeans Platform. The details of the menu items, toolbar menus and services added by the Flow Modeller application modules are described into each section. Here we provide a common overview of where these elements are added in order to be able to understand where to look for and how to change them if needed. Every modules defines a **layer.xml** that describes a virtual file system where you can have files and folders. This layers are merged together to make the composed view that define the overall application. The application menu, for example, is build from the content of the **Menu** folder: each sub folder is a menu in the main menu (for example the folder **Menu/File** creates a menu **File** in the main menu). The folder merged content, that can be either a file or a folder, specify what actions of sub menu are presented in the user interface. Actions are commonly defined in the **Actions** folder and then referenced in the **Menu** sub folders. In this way the very same action can be also put in the toolbar, under the **Toolbars** folder. Furthermore, in the **Shortcuts** folder you can define a key combination to activate an action, using a reference to the instance declared in the **Actions** folder.

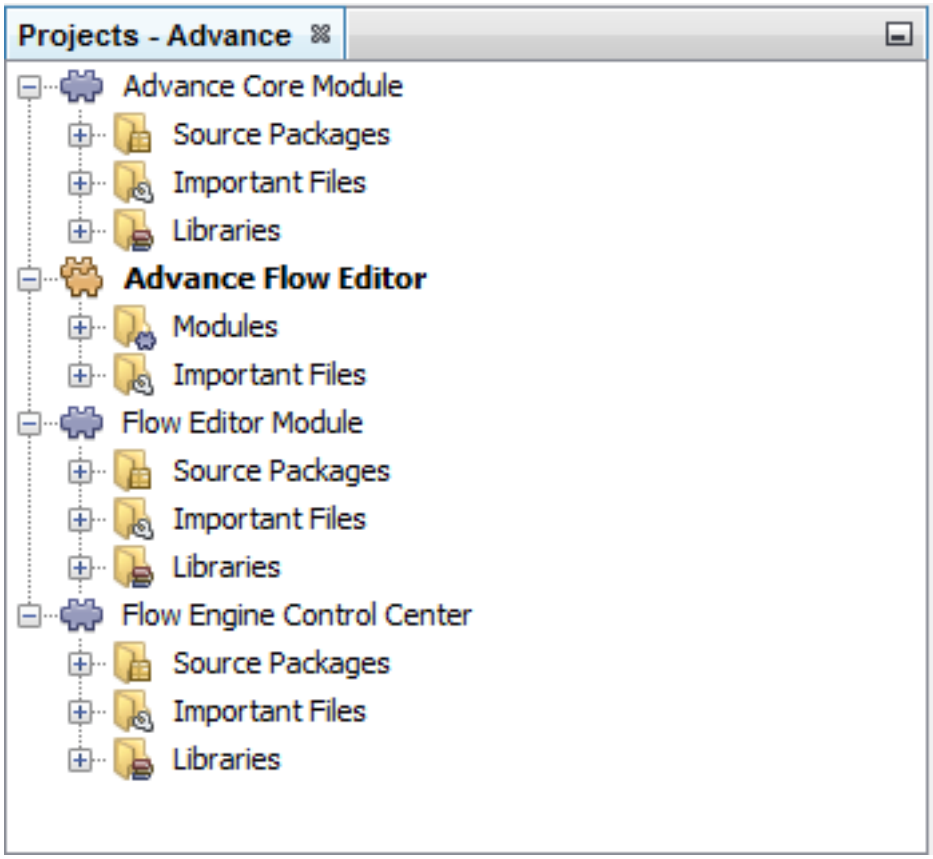

Figure 4: The Flow Editor Module structure in NetBeans

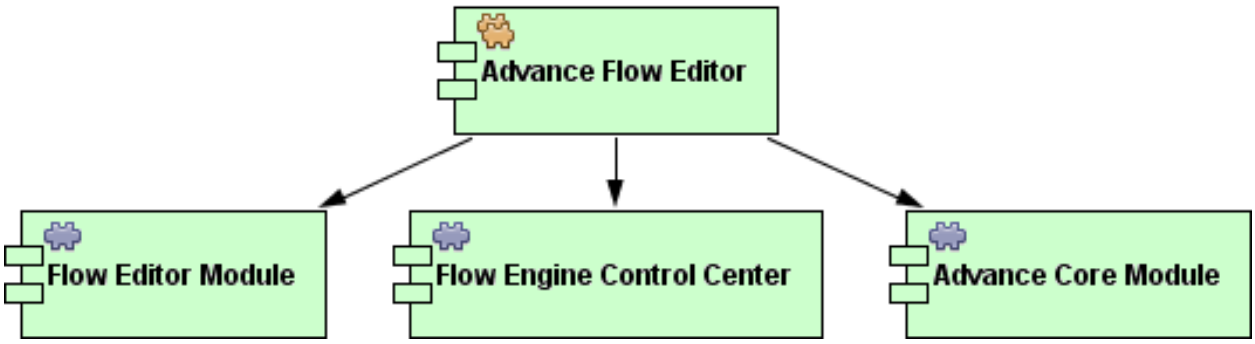

Figure 5: The Flow Editor Module structure

3.2 Modules

3.2.1 ADVANCE Core Module

The Advance Core Module is a library wrapper modules, i.e. it provides a compatibility layer between the NetBeans module system and legacy library JARs. We use this module to provide access to the Advance Engine JAR as well as to the third party libraries needed by it. As a consequence the public API of this module is composed of all the classes found in the wrapped JARs.

Public API	All wrapped classes
Private packages	None. This module is just a wrapper for external libraries

3.2.2 Flow Editor Module

This is the main module that provides the editing capabilities to the Flow Modeller application. It is composed of several packages and provides services to detect the flow description XML format, create and visually edit flow descriptions. The module depends on the **Advance Core Module** described in the previous section and on other several standard modules as shown in Figure 6.

3.2.2.1 Public API

eu.advance.logistics.flow.editor.model defines the shared observable model. Any code interested in changes to the model can register as a listener to be notified. It mostly reflects the flow description model of the engine but adds featured required by an editor like a navigable hierarchy, change events, visual display, error status.

3.2.2.2 Classes and interfaces

- class `AbstractBlock`
- class `BlockBind`
- class `BlockCategory`
- class `BlockParameter`
- class `CompositeBlock`
- class `ConstantBlock`
- class `FlowDescription`

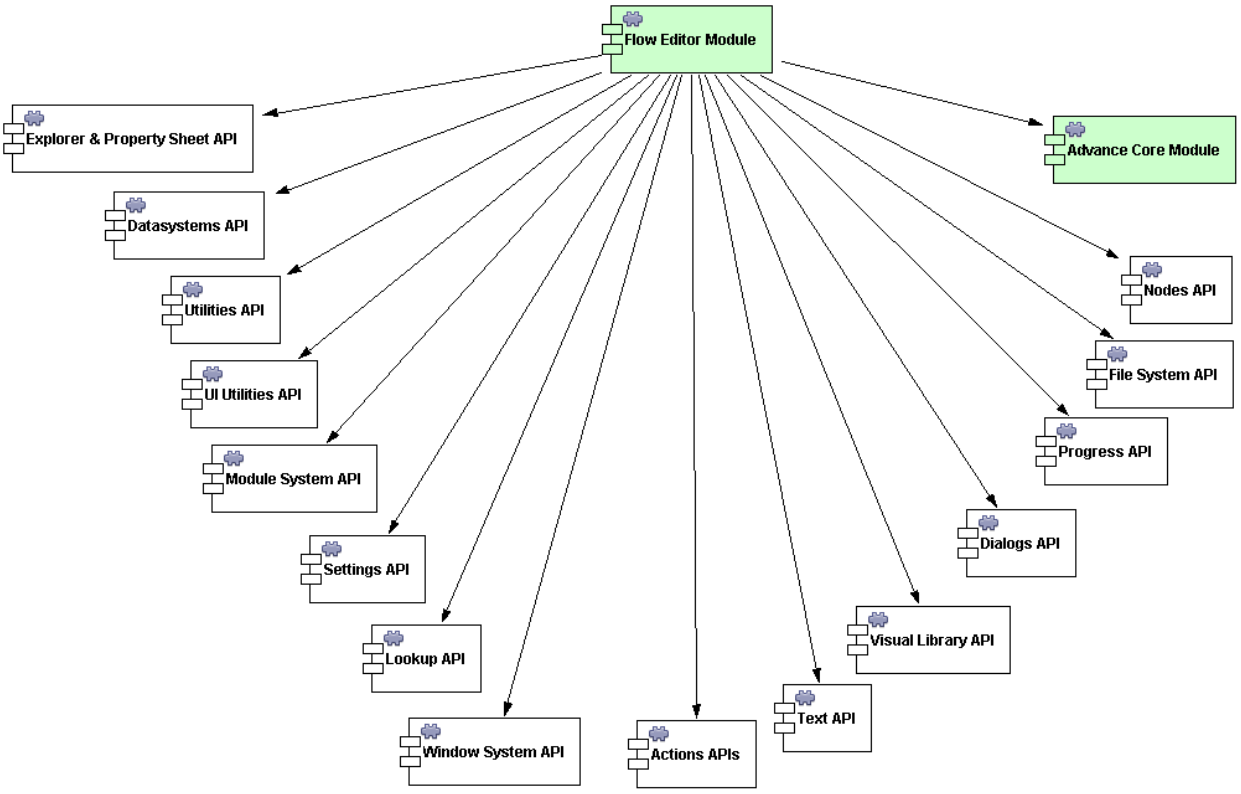

Figure 6: Standard sub-modules of the NetBeans Platform

- class FlowDescriptionIO
- interface FlowDescriptionListener
- class SimpleBlock

3.2.2.3 Enumerations

enum eu.advance.logistics.flow.editor.model.FlowDescriptionChange

- BLOCK_RENAMED: the block ID has changed.
- BLOCK_MOVED: the block position in the graph has changed.
- SIMPLE_BLOCK_ADDED: a simple block has been added.
- SIMPLE_BLOCK_REMOVED: a simple block has been removed.
- COMPOSITE_BLOCK_ADDED: a composite block has been added.
- COMPOSITE_BLOCK_REMOVED: a composite block has been removed.
- CONSTANT_BLOCK_ADDED: a constant block has been added.
- CONSTANT_BLOCK_REMOVED: a constant block has been removed.
- CONSTANT_BLOCK_CHANGED: a constant block value has changed.
- ACTIVE_COMPOSITE_BLOCK_CHANGED: the active composite block (i.e. the composite block currently visualized) has changed.
- BIND_CREATED: a new bind has been created.
- BIND_REMOVED: a bind has been removed.
- BIND_ERROR_MESSAGE: a new error message for a bind.
- PARAMETER_CREATED: a new parameter has been added to a block.
- PARAMETER_REMOVED: an existing parameter has been removed from a block.
- PARAMETER_RENAMED: a parameter ID has changed.
- PARAMETER_CHANGED: an parameter has changed type.
- COMPILATION_RESULT: a new compilation result is available.
- SAVING: the flow description is about to be saved to disk.
- CLOSED: the flow description document has been closed.

3.2.2.4 Private packages

- **eu.advance.logistics.flow.editor**: it contains the layer definition, the top panels (visual editor, navigator, tree view, palette), the *new* and *open* actions, XML flow description file type support, XML block description file type support, clipboard support, window layout.
- **eu.advance.logistics.flow.editor.actions**: it contains actions classes (which implements the javax.swing.Action interface) and support classes (like dialog boxes used to get input from the user). As this package contains user visible GUI elements, there is a Bundle.properties file where the text messages can be internationalized in different languages.
- **eu.advance.logistics.flow.editor.diagram**: it contains classes related to the Visual Library used to visualize the flow diagrams. There are classes to define custom Widgets in both appearance and behaviour; there is a custom colour scheme to fit with the project's colours, like shades of green; there are classes that handle the drag & drop of elements from the palette.
- **eu.advance.logistics.flow.editor.images**: it contains icon resources used in the module.
- **eu.advance.logistics.flow.editor.palette**: it defines the tree hierarchy used in the palette component to visualize the blocks catalogue.
- **eu.advance.logistics.flow.editor.palette.images**: it contains icon resources used in the palette to represent block categories.
- **eu.advance.logistics.flow.editor.tree**: it contains the tree hierarchy that represent the flow description in the navigator view.
- **eu.advance.logistics.flow.editor.undo**: each class of this package represent all the possible changes to the model that can be done (and thus undone).
- **eu.advance.logistics.flow.editor.util**: it contains utility methods.

3.3 Control Center Module

The Flow Editor Control Center module provides the connection between the graphical editor and the Engine Control Center. It add a new menu to the application with items to connect to a remote engine, upload, download, debug and verify a flow description, manage realms. It also adds a button to the toolbar to verify the currently open flow description. This module depends on the Advance Core Module, that provides the Engine JAR, and to the Flow Editor Module, that provides the model classes to access the flow description opened in the graphical editor.

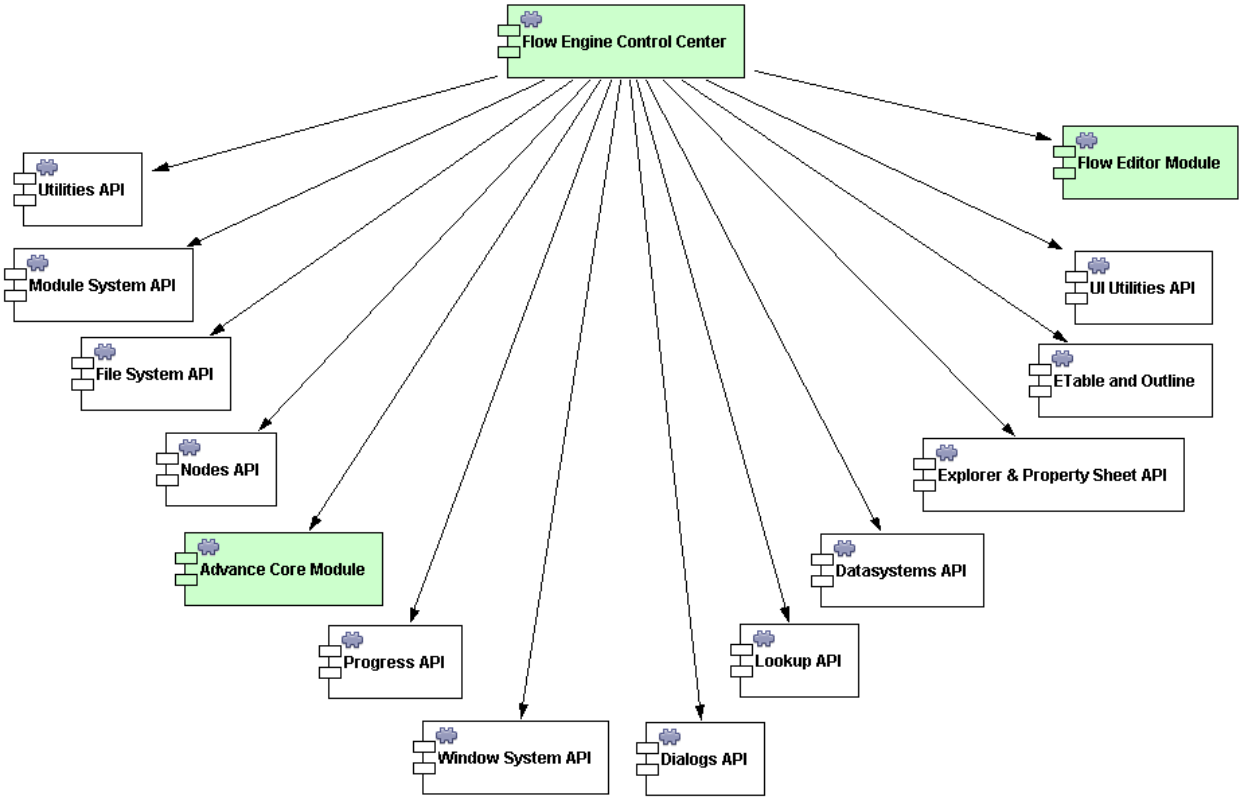

Figure 7: Control Center module dependencies

Public API	None. This module is not designed to be extended.
Private packages	**eu.advance.logistics.flow.engine.controlcenter**: It contains the action classes that implement the functionalities to connect and operate with the Engine Control Center. It also contains the Bundle.properties file with the labels and string used to visualize text in the application.

3.4 Class documentation

3.4.1 AbstractBlock

`eu.advance.logistics.flow.editor.model.AbstractBlock`

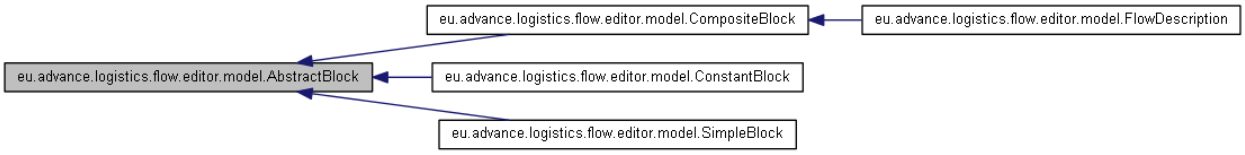

Figure 8: Inheritance diagram of AbstractBlock

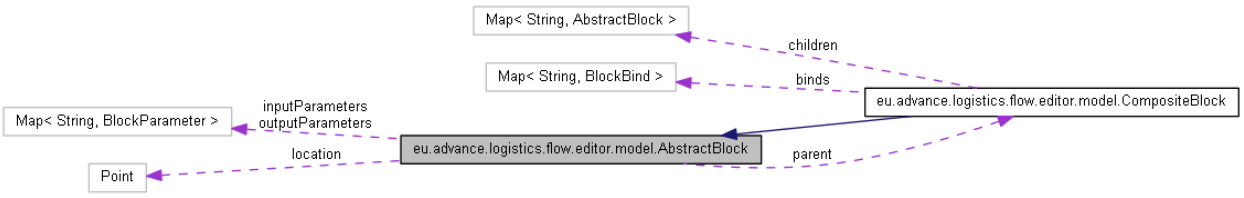

Figure 9: Collaboration diagram of AbstractBlock

Members:

- `String getId()`
  Return the block unique ID.
- `boolean setId(String id)`
  Set the unique ID of this block, checking if the specified ID is valid (i.e. there are no duplicates in the flow description). Returns true if the new ID has been applied.
- `Collection<BlockParameter> getInputs()`
  Return the list of input parameters for this block.
- `Collection<BlockParameter> getOutputs()`
  Return the list of output parameters for this block.

- `void addParameter(BlockParameter param)`
  Add a new parameter to this block. The parameter is described by the methodâ€™s argument.
- `void removeParameter(BlockParameter param)`
  Remove the specified parameter from this block.
- `BlockParameter getInputOrOutputParameter(String paramId)`
  Return a parameter with the specified ID. It searches first in the input parameters and then in the output ones.
- `CompositeBlock getParent()`
  Return the parent composite block, i.e. the composite block containing this block, or null if the block is a root element (i.e. a flow description).
- `void setParent(CompositeBlock parent)`
  Set a new block as the parent of this one.
- `int compareTo(AbstractBlock other)`
  Compare two blocks using theirs IDs. Returns the result of the string comparison between this.ID and other.ID.
- `void setLocation(Point point)`
  Set a new location for the position of the block in the graphical view.
- `Point getLocation()`
  Return the current 2D location in the graphical view of the flow or composite block.
- `void setTooltip(String tt)`
  Set the tooltip to display.
- `String getTooltip()`
  Return the tooltip to display.
- `abstract AbstractBlock createClone(CompositeBlock newParent)`
  Implemented in `CompositeBlock`, `ConstantBlock` and `SimpleBlock`.
- `BlockParameter createInput(AdvanceBlockParameterDescription desc)`
  Create a new input parameter using the provided description, adds it to this block and returns the new instance.
- `BlockParameter createOutput(AdvanceBlockParameterDescription desc)`
  Create a new output parameter using the provided description, adds it to this block and returns the new instance.
- `abstract void destroy()`
  Implemented in CompositeBlock, ConstantBlock and SimpleBlock.

- `List<BlockBind> getActiveBinds()`
  Return the list of currently active binds.
- `FlowDescription getFlowDiagram()`
  Return the flow descriptor containing this block. Reimplemented in `FlowDescription`.

3.4.2 BlockBind

`eu.advance.logistics.flow.editor.model.BlockBind`

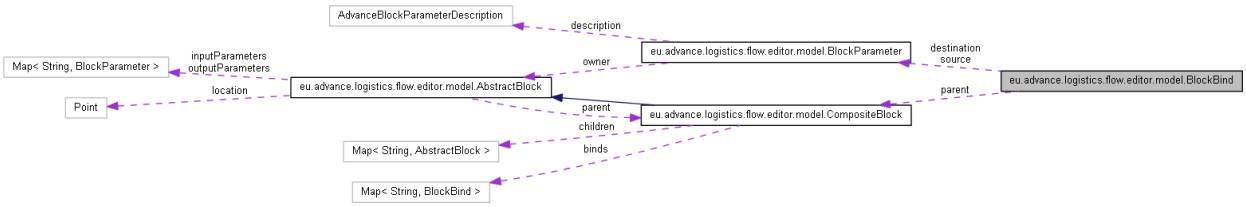

Figure 10: Collaboration diagram of BlockBind

Members:

- `String id` The unique identifier of the wire.
- `final BlockParameter source` The source block reference.
- `final BlockParameter destination` The destination block reference.
- `BlockBind(CompositeBlock parent, String id, BlockParameter src, BlockParameter dst)`
  Create a new BlockBind taking the specified values for: parent block, ID of the new bind, source parameter, destination parameter.
- `int compareTo(BlockBind other)`
  Compare two binds using their full ID build from the parent ID and the bind ID. Return the result of the string comparison of such full Ids (parent.ID + "." + bind.ID).
- `void destroy()`
  Perform all actions required to destroy a bind, freeing any used resources.
- `boolean equals(BlockParameter src, BlockParameter dst)`
  Compare two binds using their full ID build from the parent ID and the bind ID. Return the result of the string equality of such full Ids (parent.ID + "." + bind.ID).
- `String getErrorMessage()`
  Return the error message associated to this bind. Error messages are generated by validation during the compilation of the flow description.

- `FlowDescription getFlowDescription()`
  Return the instance of the flow description containing this bind.
- `CompositeBlock getParent()`
  Return the parent block containing this bind.
- `void setErrorMessage(String errorMessage)`
  Set a new error message associated to this bind. Error messages are generated by validation during the compilation of the flow description.

3.4.3 BlockCategory

`eu.advance.logistics.flow.editor.model.BlockCategory`

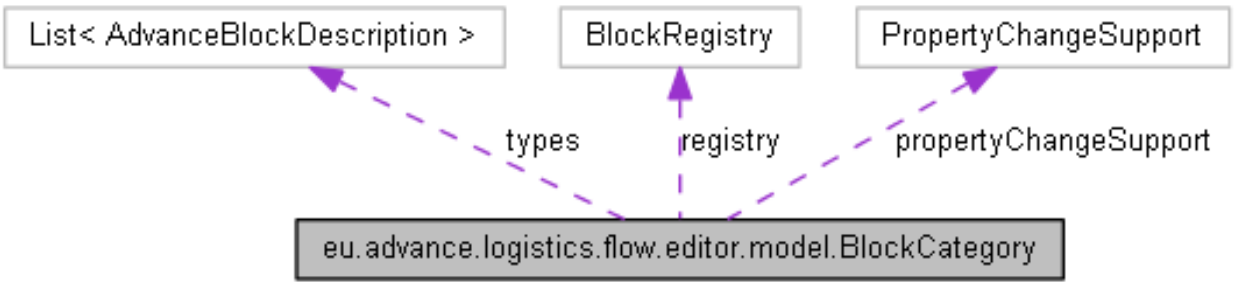

Figure 11: Collaboration diagram of BlockCategory

Members:

- `BlockCategory(BlockRegistry registry, String id, String name, String imageUrl)`
  Create a new block category using the reference to the block catalog registry, the ID of the category, the display name and a URL to the image representing the category icon.
- `void addPropertyChangeListener(PropertyChangeListener listener)`
  Add a new PropertyChangeListener to listen for events.
- `void addType(AdvanceBlockDescription type)`
  Add a new block type to this category.
- `void addType(int index, AdvanceBlockDescription type)`
  Add a new block type to this category in the specified position.
- `int compareTo(BlockCategory o)`
  Compare two categories using their IDs.

- `boolean equals(Object other)`
  Check for equality using the categoryâ€™s ID.
- `String getId()`
  Get the category unique ID.
- `String getImage()`
  Get the URL to the category icon.
- `Image getImageObject()`
  Get the image object of the category icon. The image is cached.
- `String getImagePath()`
  Get the full path of the category icon.
- `String getName()`
  Gets the display name for this category.
- `List<AdvanceBlockDescription> getTypes()`
  Get a list of block types contained in this category.
- `void removePropertyChangeListener(PropertyChangeListener listener)`
  Remove a PropertyChangeListener.
- `void removeType(AdvanceBlockDescription type)`
  Remove a block type from this category.

3.4.4 BlockParameter

`eu.advance.logistics.flow.editor.model.BlockParameter`

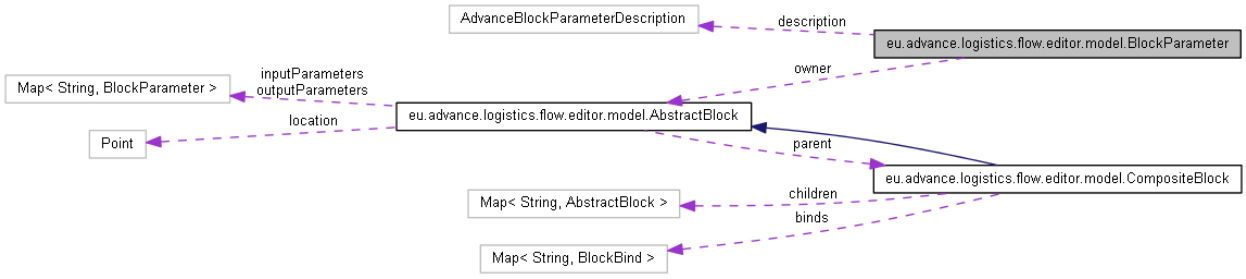

Figure 12: Collaboration diagram of BlockParameter

Members:

- `enum Type{ INPUT, OUTPUT }`

- `BlockParameter(AbstractBlock parent, AdvanceBlockParameterDescription desc, Type type)`
  Create a new block parameter with the specified parent (i.e. containing blokc), parameter description and type (i.e. input or output).
- `boolean canChangeId(String id)`
  Return true if the parameter can change its ID to the one specified. This methods checks if the new ID is unique inside the list of parameters of the parent block.
- `int compareTo(BlockParameter o)`
  Compare two parameter using their IDs.
- `BlockParameter createClone(AbstractBlock parent)`
  Create a new instance of the parameter with the same characteristics of this parameter and a specified parent block.
- `void destroy()`
  Destory this parameter and frees any used resources.
- `AdvanceBlockParameterDescription getDescription()`
  Get the description of this parameter.
- `String getDisplayName()`
  Get the display name of this parameter.
- `FlowDescription getFlowDescription()`
  Get the flow description containing this parameter.
- `String getId()`
  Get the unique ID of this parameter.
- `AbstractBlock getOwner()`
  Get the owner of this parameter, i.e. the block containing it.
- `String getPath()`
  Get the full path of this parameter built as owner.ID + "." + this.ID.
- `void setDescription(AdvanceBlockParameterDescription description)`
  Set a new description for this parameter.
- `void setId(String id)`
  Set a new ID for this parameter. Caller must check that canChangeId returns true.

3.4.5 CompositeBlock

`eu.advance.logistics.flow.editor.model.CompositeBlock`

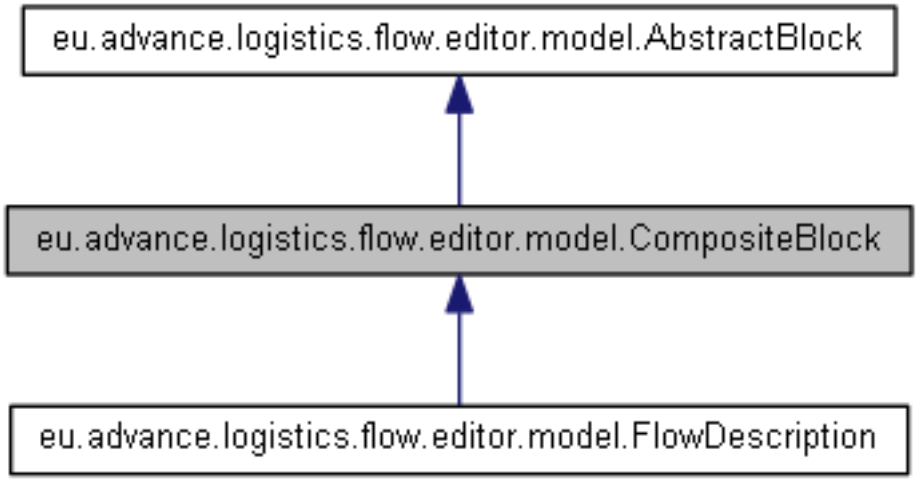

Figure 13: Inheritance diagram of CompositeBlock

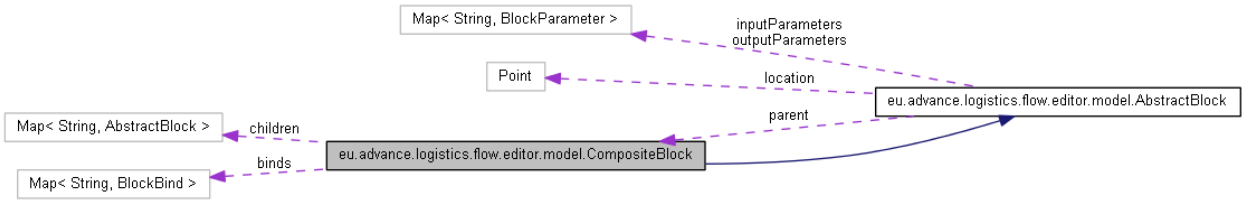

Figure 14: Collaboration diagram of CompositeBlock

Members:

- `CompositeBlock(String id)`
  Create a new composite block with the specified ID.
- `void addBind(BlockBind c)`
  Add the specified bind to this composite block.
- `void addBlock(SimpleBlock block)`
  Add the specified simple block to this composite block.
- `void addComposite(CompositeBlock block)`
  Add the specified composite block to this composite block.

- `void addConstant(ConstantBlock block)`
  Add the specified constant block to this composite block.
- `BlockBind createBind(BlockParameter src, BlockParameter dst)`
  Create a new bind from the source parameter to the destination parameter. Return a new instance of the class representing the bind.
- `BlockBind createBind(String bindId, BlockParameter src, BlockParameter dst)`
  Create a new bind from the source parameter to the destination parameter and using the specified ID. Return a new instance of the class representing the bind.
- `SimpleBlock createBlock(AdvanceBlockDescription desc)`
  Create a new simple block using the specified block description.
- `SimpleBlock createBlock(String blockId, AdvanceBlockDescription desc)`
  Create a new simple block, inside this composite block, using the specified block ID and description.
- `CompositeBlock createClone(CompositeBlock newParent)`
  Create a new copy of this simple block, inside this composite block, using the specified parent. Implements `AbstractBlock`.
- `CompositeBlock createComposite()`
  Create a new composite block inside this composite block.
- `CompositeBlock createComposite(String blockId)`
  Create a new composite block inside this composite block and using the supplied block ID.
- `ConstantBlock createConstant()`
  Create a new constant block.
- `ConstantBlock createConstant(String blockId)`
  Create a new constant block using the specified ID.
- `void destroy()`
  Destroy this composite block and free nay used resources. Implements `AbstractBlock`.
- `BlockBind find(BlockParameter src, BlockParameter dst)`
  Find a bind that connect the source with the destination parameters, return null if none is found.
- `BlockParameter findBlockParameter(String blockId, String paramId)`
  Find a parameter with the specified ID contained in a block specified by the block ID.
- `String generateConstantId()`
  Generate a new ID for a constant block.

- `List<BlockBind> getActiveBinds(AbstractBlock child)`
  Get the list of active binds for the specified block, i.e. all the binds incoming or outgoing from the block.
- `Collection<BlockBind> getBinds()`
  Get the collection of bind contained in this composite block.
- `Collection<AbstractBlock> getChildren()`
  Get the collection of blocks contained in this composite block.
- `void removeBind(BlockBind bind)`
  Remove the specified bind from this composite block.
- `void removeBlock(SimpleBlock block)`
  Remove the specified simple block from this composite block.
- `void removeComposite(CompositeBlock block)`
  Remove the specified composite block from this composite block.
- `void removeConstant(ConstantBlock block)`
  Remove the specified constant block from this composite block.

3.4.6 ConstantBlock

`eu.advance.logistics.flow.editor.model.ConstantBlock`

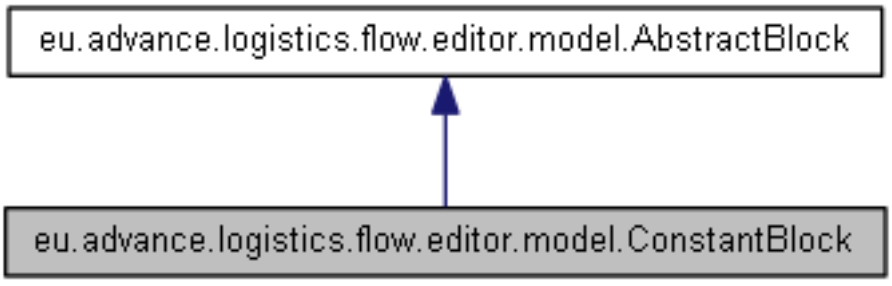

Figure 15: Inheritance diagram of ConstantBlock

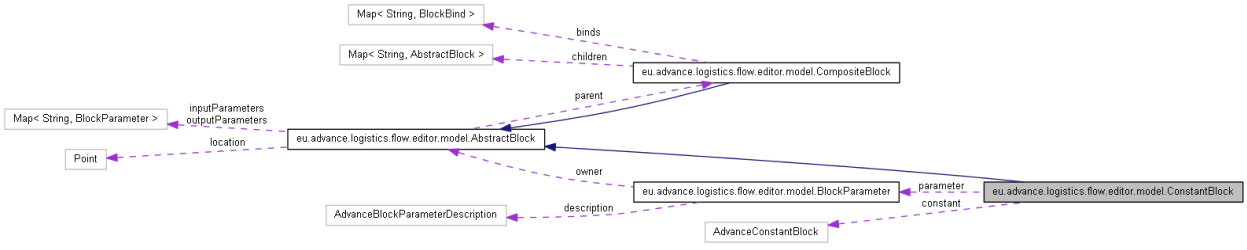

Figure 16: Collaboration diagram of ConstantBlock

Members:

- `static final String DEFAULT_PARAMETER_NAME = "constant"`
- `ConstantBlock(String id)`
  Create a new constant block with the specified ID.
- `ConstantBlock createClone(CompositeBlock newParent)`
  Create a complete copy of this constant block using the specified parent block. Implements `AbstractBlock`.
- `void destroy()`
  Destroy this constant block freeing all the resources used. Implements `AbstractBlock`.
- `AdvanceConstantBlock getConstant()`
  Get the description of the constant block.
- `BlockParameter getParameter()`
  Get the parameter of the constant block.
- `String getTypeAsString()`
  Get the textual representation of the type of this constant block.
- `String getValueAsString()`
  Get the textual representation of the value of this constant block.
- `void setConstant(AdvanceConstantBlock constant)`
  Set the description of this constant block.

3.4.7 FlowDescription

`eu.advance.logistics.flow.editor.model.FlowDescription`

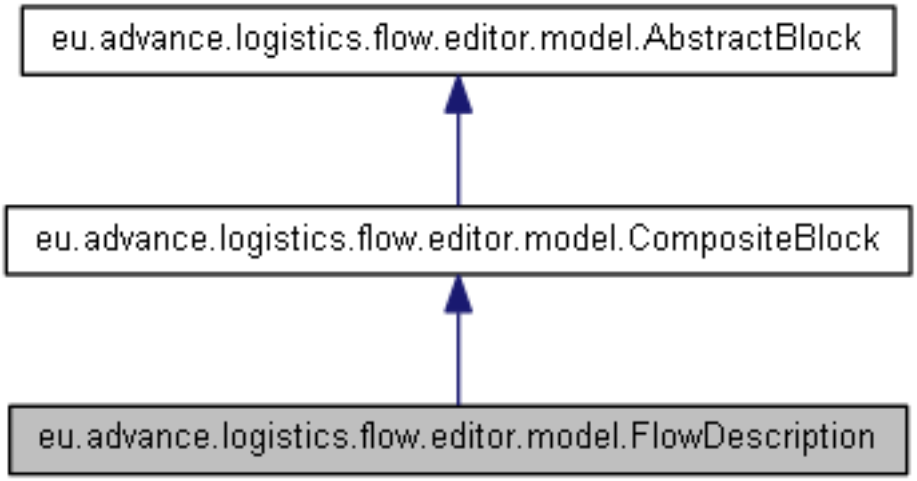

Figure 17: Inheritance diagram of FlowDescription

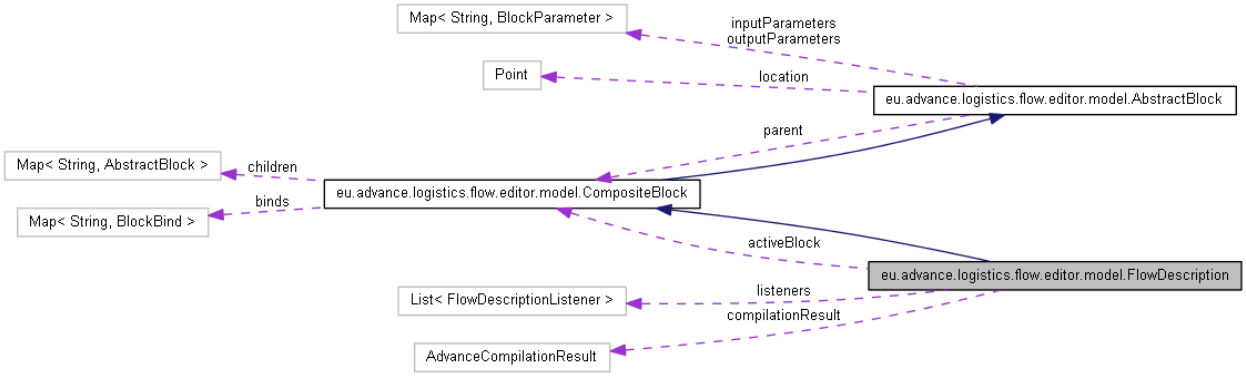

Figure 18: Collaboration diagram of FlowDescription

Members:

- `FlowDescription(String id)`
  Create a new flow description with the specified ID.
- `void addListener(FlowDescriptionListener l)`
  Add a new listener for changes to the flow description.
- `AdvanceCompositeBlock build()`
  Build the representation of the flow description using the Advance Engine model.
- `static FlowDescription create(AdvanceCompositeBlock compositeBlock)`
  Create a flow description from the Advance Engine model.

- `void fire(FlowDescriptionChange event, Object... params)`
  Fire a new event to all the registered listeners.
- `CompositeBlock getActiveBlock()`
  Return the currently active composite block, i.e. the block currently open in the graphical editor.
- `AdvanceCompilationResult getCompilationResult()`
  Return the last compilation result.
- `FlowDescription getFlowDiagram()`
  Return a reference to itself. Reimplemented `AbstractBlock`.
- `static AdvanceCompositeBlock load(InputStream s)`
  `tabthrows IOException`
  Load a flow description from an input stream.
- `void removeListener(FlowDescriptionListener l)`
  Remove the specified listeners.
- `static void save(OutputStream s,`
  `AdvanceCompositeBlock compositeBlock) throws IOException`
  Save a flow description to an output stream.
- `void setActiveBlock(CompositeBlock activeBlock)`
  Set the currently active block, i.e. the block that is open in the graphical view.
- `void setCompilationResult(AdvanceCompilationResult result)`
  Set the compilation result as returned by the Engine.

3.4.8 FlowDescriptionListener

`eu.advance.logistics.flow.editor.model.FlowDescriptionListener`

- `void flowDescriptionChanged(FlowDescriptionChange event,`
  `Object... params)`
  Called to notify that a new event has happened. The parameters change based on the type of the event.

3.4.9 SimpleBlock

`eu.advance.logistics.flow.editor.model.SimpleBlock`

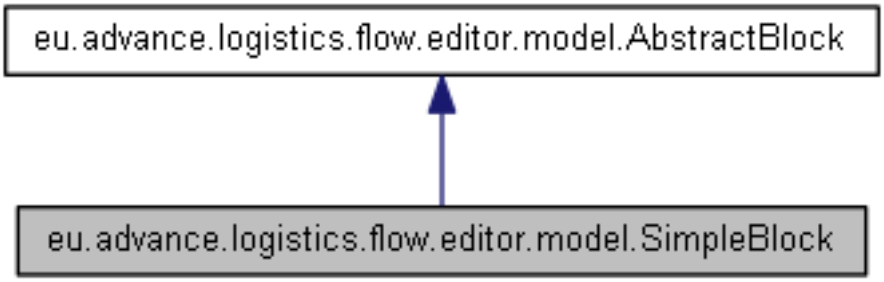

Figure 19: Inheritance diagram of SimpleBlock

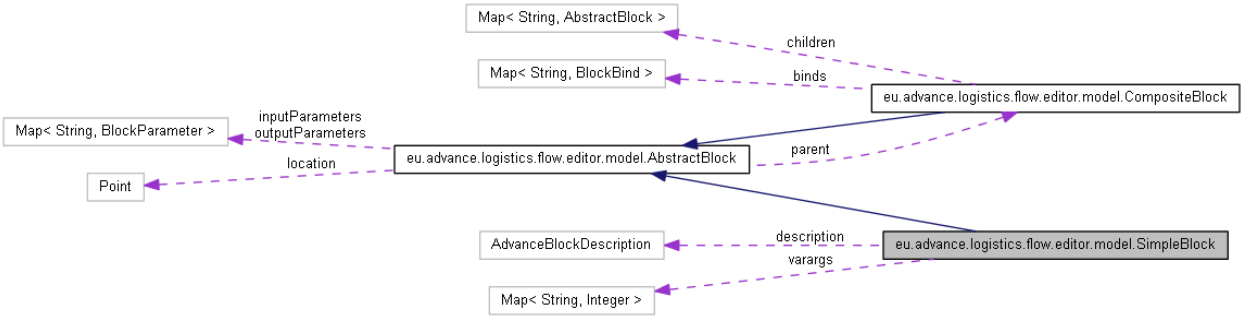

Figure 20: Collaboration diagram of SimpleBlock

Members:

- `final AdvanceBlockDescription description` Reference to the block's definition.
- `Map<String, Integer> varargs` Keeps track of the number of various variable-length arguments.
- `SimpleBlock(String id, AdvanceBlockDescription desc)`
  Create a new simple block using the specified ID and description.
- `SimpleBlock createClone(CompositeBlock newParent)`
  Create a new instance of this simple block with the same parameters and a new parent composite block. Implements `AbstractBlock`.
- `void destroy()`
  Destroy this simple block and free all the used resources. Implements `AbstractBlock`.

4 ADVANCE Elicitation Tool

4.1 Overview

The Advance Elicitation Tool is a web app developed using Java Server Pages (JSP), Java Servlet and JavaScript technologies. The web app is compiled into a WAR (Web application ARchive) and can be deployed onto any web server with a servlet container, such as Apache Tomcat or Jetty. The default binary distribution is tested using Apache Tomcat versions 6 and 7.

4.1.1 Pages

The front end of the web app is composed of several pages described below:

4.1.1.1 Login page – index.jsp

Initial page that allows the user to log in the application. It requires a valid username and password.

4.1.1.2 File list page – fileList.jsp

Page that shows the list of uploaded files and let the user interact with them.

The list shows every files that has been previously uploaded into the server. The files that are under manipulation by other users can't be chosen. The list is updated both automatically, every 15 seconds, and via a button that can be clicked by the user. The page provides the user 4 possible actions to perform against a file: download it, edit it, view it, delete it. The page also gives the possibility to upload a new valid XML file into the server. Eventually the page contains a link to a PDF quick guide of the whole web app to assists the user with its usage.

4.1.1.3 Edit page – editFile.jsp

Page that show and allow o edit the structure of an XML file selected into the file list page.

The page is split into 2 sides: the left side reproduces the XML structure of the file as a list of expandable and selectable nodes. the right side shows the content of the XML node that has been selected into the left side.

The bottom part of the left side contains 3 buttons that allow the user to: reload the XML structure, save the modifications that has been done, and to returns to the file list page.

4.1.1.4 View page – viewFile.jsp

Page that gives a graphic representations of the XML structure of the files chosen into the file list page. The page let the user click on a node and shows the path that link it to the file root node.

Figure 21: Elicitation tool pages

4.2 Page layout

All the pages share a common HTML layout described. The main components of this layout are HTML's *div* elements that acts as content containers. Each element is described below:

Name	ID	Description
Wrapper	wrapper	Content wrapper
Page header	page-header	Contains the web app's title and the project's logo
Page content	content	Contains the effective dynamic portion of every page
Page Footer	footer	a static resource included in every page that contains the copy-right notes.

4.2.1 Common CSS rules

Every page imports a common CSS style sheet file named *general.css* that sets a common rule set to properly display the portions of HTML structures that are present into every page.

- The applied box model is the border-box model.
  `https://developer.mozilla.org/en-US/docs/Web/CSS/box-sizing`

- The used font is an external font, provided by Google Fonts web tool. `http://www.google.com/fonts/specimen/Open+Sans`
- The font-size is determined through the *62.5%* technique. `https://developer.mozilla.org/en-US/docs/Web/CSS/font-size#Ems`
- The main colors chosen for the web pages are: green (#388d3c, #317834), gray (#c0c0c0) and white (#ffffff).
- The green is applied, with a CCS3 linear gradient, to all the buttons and to the element with class equals to *page-title* (a sub-header which content changes in every page).
- The gray is applied to the static header and the static footer included into every pages.
- The white is used applied to the wrapper div, to the content div and to the main-content div. The visual result is that the area where the dynamic content is displayed is white.
- The layout could be named as *semi-fixed*. It varies from a minimum width of 60em to a maximum width of 96em. The inner element's width is defined with use of percentages.

4.2.2 Common JavaScript files

The whole web app relies deeply on JavaScript technologies. In order to ease the development and to reduce the size of generated JS files the jQuery framework (`http://jquery.com/`) is adopted. Every page's script makes use of this framework. There are also other external libraries used on specific page (for instance jQuery UI and d3 version 2).

4.3 Screens

The following subsections describe the pages in depth.

4.3.1 Login page

This page shows a basic login dialog box where the user can insert an username and a password. When both of them are valid it allows the user to access the web-app and make use of the web app. The login process seeks the provided data into an XML file (located into the folder named *user*) that contains the list of allowed users. When the provided data (username and password) both match an XML node a Java Bean that represent the user's data is instantiated and stored into the HTTPSession, in order to let the engine be able to recognize the user among every HTTPRequest. This java Bean, called userBean, will also store all the information about the user's permissions. The user's name will be added to a list that track the name of all the user that are using the web app.

Below is a template for the XML allowed-user list.

```
<users>
  <user name="johndoe" password="123">
    <permission type="edit" allow="true"/>
    <permission type="view" allow="true"/>
    <permission type="upload" allow="true"/>
    <permission type="download" allow="true"/>
    <permission type="delete" allow="true"/>
  </user>
  <user name="johndoealterego" password="321">
    <permission type="edit" allow="false"/>
    <permission type="view" allow="false"/>
    <permission type="upload" allow="false"/>
    <permission type="download" allow="false"/>
    <permission type="delete" allow="false"/>
  </user>
<users>
```

4.3.1.1 Permissions

The permissions coincide with the five actions that the web app provides:

- **edit** an existing file,
- **view** its horizontal tree view,
- **download** an existing file,
- **delete** an existing file,
- **upload** a new *Galatea* XML file.

These permissions are assigned for each user into the XML allowed-user list as children nodes with a Boolean value that indicate whether they have this permission or not.

4.3.2 File list

Once the login process has been successfully executed the user will be landed to this page. This page is cockpit of the web app that let the user perform all the actions they need to do (according to their permissions). The page shows the list of uploaded file. The file that are under editing by other users are locked otherwise the file can be selected by clicking on its name.

This list is automatically updated via an AJAX request that receive an update full file list with the eventual user's name that is currently editing the file and update the DOM accordingly to it.

The list can also be manually updated from the user by clicking the update button in the right side of the top bar beneath the header.

Once a file is selected the user can perform 4 actions against it by clicking the related button:

- **Download** will download the selected file
- **Edit** will redirect the user to the page that let him edit the selected file
- **View** will redirect the user to the page that shows the horizontal tree view of the selected file
- **Delete** will delete the selected file

The user can also click on the 5th button, that is **Upload**, which will open a modal dialog box that let they choose a file from the hard disk and upload it. This process will also evaluate and validate the XML file according to the specification of the *Galatea* XML as stated in *D6.4 Algorithms to extract expert cognitive models, decision driver factors and decision classes*. If the file is successfully validated it will be uploaded to the file repository (a folder on the server) and will be shown in the file list.

The redirection to the other pages (view and edit) will be performed via a form tag through a post request. The form is placed into the bottom toolbar and its action attribute will be manipulated via JavaScript to correctly redirect the user to the appropriate page. The form give the possibility to send the reference of the selected file to the redirection page that otherwise wouldn't be able to recognize it.

Clicking on the question mark beside the button that update the file list the user has the possibility to view an online quick reference guide.

4.3.3 Edit page

This page let an user perform the editing action against the file that they have previously selected on the file list page. The page is split in two sections, the XML tree section and the Node editor section, described in the following paragraphs.

4.3.3.1 XML Tree section

This side is populated by a JavaScript function that retrieve through AJAX the selected file, parse it recursively and create an HTML representation of the XML structure as a typical tree view with expandable nodes, totally similar to Windows Explorer. Thus, every node except the root can be selected by clicking on it, when it has children nodes it can be expanded by clicking on a plus symbol located on its left side and, after having been expanded its children nodes can be hidden by clicking on the same symbol, which became a minus symbol.

4.3.3.2 Node editor section

This section is activated and populated by a JavaScript function every time a node is selected and, its contents vary according to the node that has been selected. To check which kind of nodes has been selected a JavaScript function will scan the selected node's children to check whether at least two of them they have RI (Relative Influence) attribute. According to these checking (that returns a Boolean value) the node editor will consist of either a **set of slider** or a **Cartesian graph**.

A **set of slider** will appear when the searching for two children with relative influence attribute is successful. One slider will be created for each node, each of them will have the value equals to the relative influence attribute value and a label corresponding to the children node's label. The user can edit either the slider value via its own handler or the label by clicking on it. Once the modification are terminated the values can be temporary stored by clicking the *Record RI* button on the bottom toolbar of the node editor section. The sliders are created by making use of jQueryUI Library (`http://jqueryui.com/slider/`) with a customized theme.

A **Cartesian** graph will be created when the same searching won't be successful and gives a graphical representation of the selected node's MG (Membership Grade) attribute values. According to the type of MG value the graph can be a single graph or a tabbed graph or, in a third case the content will consists of a new slider set. The single graph is a single Cartesian Graph, a tabbed graph will consists of a set of Cartesian graphs that can be selected by a set of tab located into the upper side of the node editor area. A graph let the user modify the value of the selected node's MG attribute value by moving the nodes placed into it and, once the editing is ended the values can be temporarily saved by clicking on the *Record MG* button. The node editor also allows the user to zoom in and out the graph and to restore to zoom to the default. The graph is created using the JSXGraph library (`http://jsxgraph.uni-bayreuth.de/wp/`).

4.3.3.3 Editor logic

The editor logic, written in JavaScript, starts from a function that acts has listen to clicks on the XML tree section's nodes, once the click is triggered and bubbles to the right DOM element, the function will execute and so the whole functions flow will start.

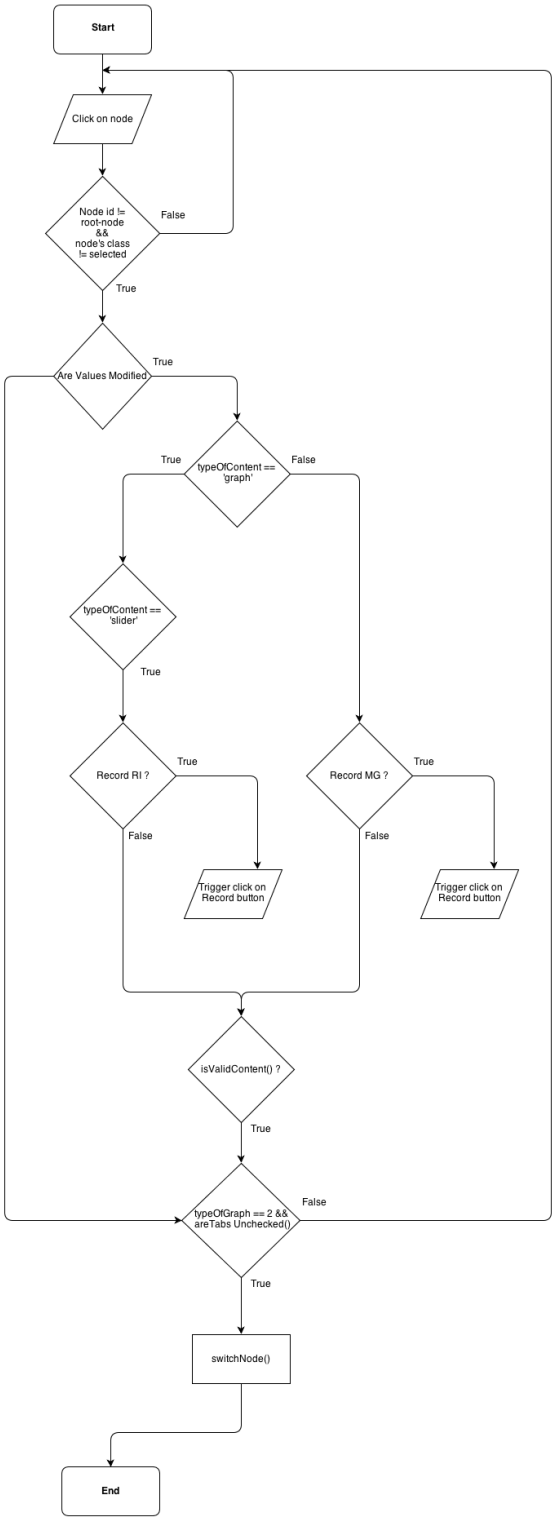

Figure 22: Elicitation Editor Logic

4.3.4 View file

This page shows the horizontal tree view of the file that was selected into the file list page. The visualization is created using the *d3* JavaScript library (`http://d3js.org/`). When the user clicks on a node into the view the path from the clicked node to the root one will be highlighted.

5 ADVANCE Live Reporter

The ADVANCE Live Reporter (ALR) is a Java 2 Enterprise Edition web application which is aimed at providing real-time evaluation capabilities to a hub-and-spoke logistics environment.

5.1 Architecture

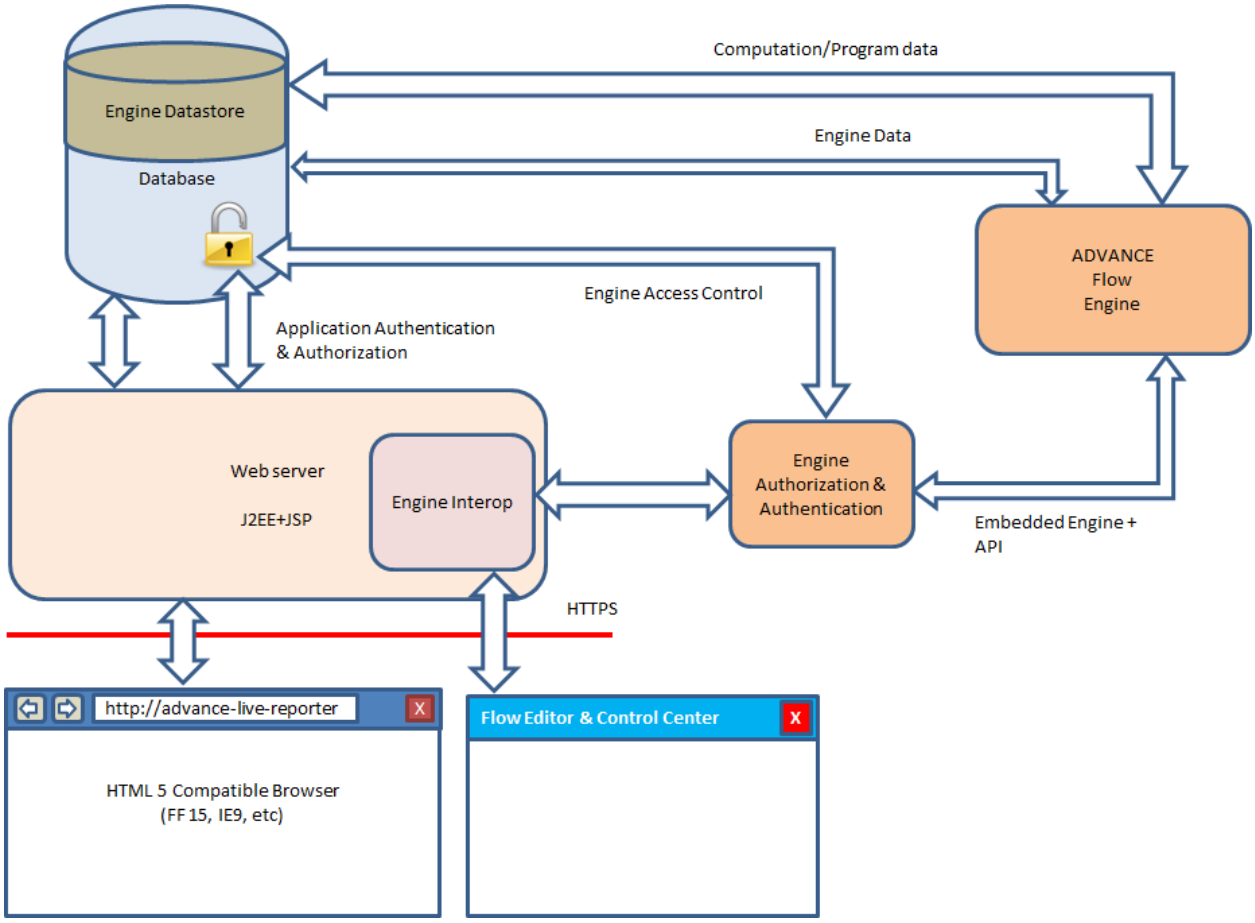

Figure 23: ALR architecture

5.2 Database design

In the following subsections, the table's primary keys are noted by underlining the relevant column names. The order in the primary key is the same as the order in the description tables.

The NUMBER type represents an integer-type, 64 bit length number. The VARCHAR type is a variable length string, usually required to be non-null (if nullness is allowed, it is noted in the

description column). The DECIMAL represents a floating point value, usually an IEEE-754 double precision value.

5.2.1 Master data tables

5.2.1.1 HUBS

Since the hub appears repeatedly in many tables and rows, using a numeric id instead a textual one saves space and improves performance. As there are usually a few hubs in the table, using SQL join is cheap or can be performed in-memory on the application side.

Name	Type	Description
id	NUMBER	The hub's unique identifier.
name	VARCHAR	The hub's display name.
width	DECIMAL	The width of the hub's premises in meters.
height	DECIMAL	The height of the hub's premises in meters.

Table 39: HUBS table

The width and height describe the rectangular size of the hub's premises. They are used by the *Scheduler* to calculate travel times within the hub and between warehouses.

5.2.1.2 DEPOTS

Depots appear repeatedly in many tables and rows, using a numeric id instead of textual ones saves space and improves performance.

Name	Type	Description
id	NUMBER	The depot's unique identifier.
name	VARCHAR	The depot's display name.

Table 40: DEPOTS table

5.2.1.3 POSTCODES

Consignments feature collection and delivery postcodes. Since the same postcodes appear in many rows, using a numeric id instead of textual ones saves space and improves performance.

Name	Type	Description
id	NUMBER	The postcode's unique identifier.
code	VARCHAR	The textual postcode.
grouping	VARCHAR	The text which groups the postcode.

Table 41: POSTCODES table

Postcodes contain a so-called `grouping` field which is used by the machine learning's timepoint aggregator to perform virtual aggregation. See 5.6.2.4 and table 42.

5.2.1.4 DEPOT_TERRITORIES

The table contains the territories of the depots at specific dates.

Name	Type	Description
hub	NUMBER	The hub identifier.
depot	NUMBER	The postcode's unique identifier.
start_date	DATE	The first date when the territory is in effect.
grouping	VARCHAR	The text which selects the groups of postcodes.

Table 42: DEPOT_TERRITORIES table

The set territory-indicating `grouping` values have the same `start_date` values. The relevance duration of a depot's territory is determined by the distinct `start_date` values (e.g., the old territory ends when a new start_date is entered). In the virtual aggregation (5.6.2.4), consignments are filtered based on their delivery postcode's grouping value is in the set of the delivery depot's territory.

5.2.2 Consignments & items

5.2.2.1 CONSIGNMENTS (& CONSIGNMENTS_HISTORY)

For performance reasons, the *CONSIGNMENTS* table contains only the most recent known consignments, whereas the *CONSIGNMENTS_HISTORY* contains all known consignments to be used in the day-by-day and during-day learning.

Name	Type	Description
id	NUMBER	The consignment's unique identifier.

Name	Type	Description
created	DATETIME	The timestamp when the consignment appeared in the system.
declared	DATETIME	The timestamp when the consignment is sure to be delivered through the network. *OPTIONAL*
hub	NUMBER	The hub identifier reference.
collection_depot	NUMBER	The collection (source) depot identifier reference.
collection_postcode	NUMBER	The collection postcode identifier reference.
delivery_depot	NUMBER	The delivery (destination) depot identifier reference.
delivery_postcode	NUMBER	The delivery postcode identifier reference.
service_level	NUMBER	The service level value. The `ServiceLevel` enum's ordinal is stored here.
item_count	NUMBER	Number of items of this consignment.
external_id	VARCHAR	The consignment's external identifier.

Table 43: CONSIGNMENTS table

5.2.2.2 ITEMS (& ITEMS_HISTORY)

For performance reasons, the *ITEMS* table contains only the items of the most recent known consignments. The *ITEMS_HISTORY* contains all items of the consignments.

Name	Type	Description
id	NUMBER	The item's unique identifier.
consignment_id	NUMBER	The parent consignment identifier reference.
external_id	VARCHAR	The item's external identifier.
height	DECIMAL	The item's height in meters.
width	DECIMAL	The item's width in meters.
length	DECIMAL	The item's length in meters.
weight	DECIMAL	The item's weight in kilograms.

Table 44: ITEMS table

5.2.3 Warehouses & layout information

5.2.3.1 WAREHOUSES

The table lists the warehouses of a hub. It also describes the warehouse's position, size and orientation relative to the hub's premises.

Name	Type	Description
hub	NUMBER	The hub identifier reference.
warehouse	VARCHAR	The warehouse's name, unique within a hub.
x	DECIMAL	The position of the warehouse within the hub's premises in meters.
y	DECIMAL	The position of the warehouse within the hub's premises in meters.
width	DECIMAL	The width of the warehouse in meters.
height	DECIMAL	The height of the warehouse in meters.
angle	DECIMAL	The angle of the warehouse, clockwise, in degrees.
forklifts	NUMBER	The maximum number of forklifts available in the warehouse.
warehouse_pair	VARCHAR	The name of another warehouse which can be considered as the twin or partner warehouse of this warehouse.

Table 45: WAREHOUSES table

The position, size and angle is interpreted as shown in Figure 24.

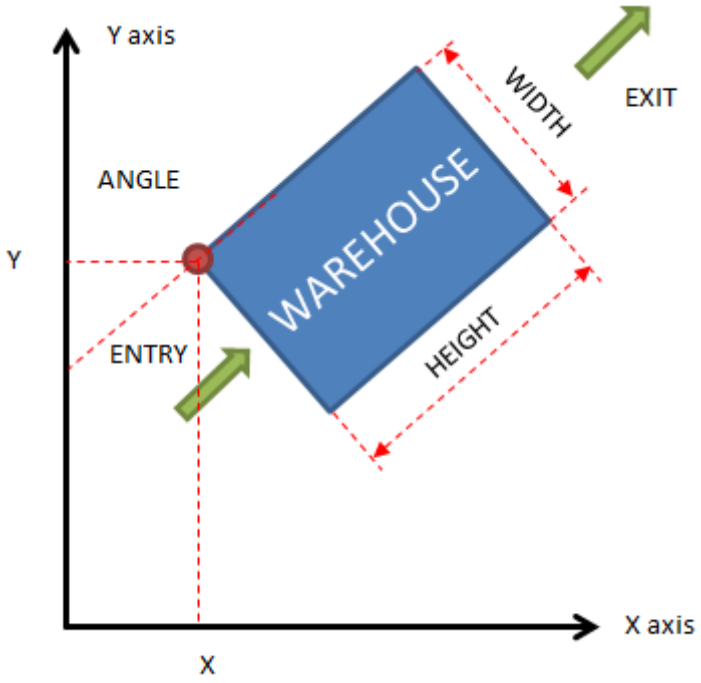

Figure 24: The explanation of the warehouse's position, size and orientation.

5.2.3.2 STORAGE_AREAS

The table describes the individual storage areas for a particular destination depot, for each warehouse.

Name	Type	Description
hub	NUMBER	The hub identifier reference.
warehouse	VARCHAR	The warehouse's name, as specified in the *WAREHOUSES* table.
side	NUMBER	Determines the storage area's location to be either on the `left = 0` or `right = 1` side of the warehouse, when looking from the entry.
idx	NUMBER	The numerical index representing a storage area.
depot	NUMBER	The destination depot of the storage area.
width	DECIMAL	The width of the storage area in meters.
height	DECIMAL	The height of the storage area in meters.
service_levels	VARCHAR	The list of service level codes (see `ServiceLevel` enum's ordinals) whose items can be stored in the storage area.
capacity	NUMBER	The logical capacity of the storage area.

Table 46: STORAGE_AREAS table

The structure permits multiple storage areas for the same depot as well as different storage areas for different service levels. The storage area's running `idx` identifies them from entry to exit direction.

The `width` and `height` are interpreted along the same dimensions as the warehouse's width and height (see Figure 24 and Figure 25).

5.2.3.3 LORRY_POSITIONS

The table describes the parking position of the lorries once they entered a particular warehouse.

Name	Type	Description
hub	NUMBER	The hub identifier reference.
warehouse	VARCHAR	The warehouse's name, as specified in the *WAREHOUSES* table.
lorry_position	NUMBER	The lorry position's index within the warehouse.
x	DECIMAL	The lorry position's X coordinate in meters.
y	DECIMAL	The lorry position's Y coordinate in meters.
width	DECIMAL	The lorry position's width in meters.
height	DECIMAL	The lorry position's height in meters.
enter_time	NUMBER	The number of seconds a lorry needs to reach this position (from entering the warehouse), on average.
leave_time	NUMBER	The number of seconds a lorry needs to leave this position (and the warehouse), on average.

Table 47: LORRY_POSITIONS table

For the interpretation of the position and size of a lorry position, please refer to Figure 25 for explanation.

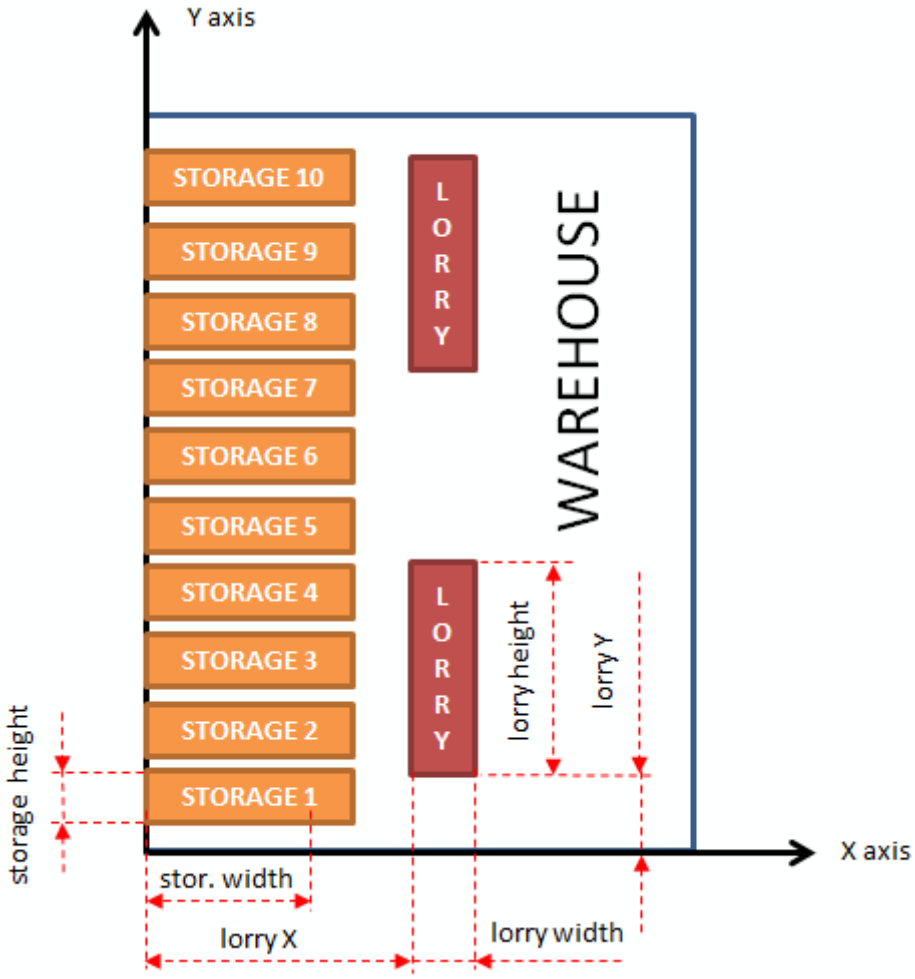

Figure 25: Storage areas and lorry position dimension interpretation

5.2.4 Events

5.2.4.1 EVENTS

The table has the global, time-ordered events list to be used by the machine learning component to perform the timepoint aggregation in a streaming fashion.

Name	Type	Description
consignment_id	NUMBER	The consignment's identifier reference, as in the *CONSIGNMENTS_HISTORY* table.
item_id	NUMBER	The item's identifier reference, as in the *ITEMS_HISTORY* table.
event_timestamp	DATETIME	The event's timestamp.
event_type	NUMBER	The event type as the `ItemEventTypes` enum's ordinal value.

Table 48: EVENTS table

Events are expected to be available on a per item basis. In case some events (for example, the creation of the consignment itself) can't be truly assigned to items, the event should be inserted for all the items of the consignment.

To support the fast, time-ordered access, the `event_timestamp` is needed to be indexed.

5.2.4.2 SCANS (& SCANS_HISTORY)

The table helps determine the state of the hub and the (possible) location of various itemss. The *SCANS* table holds only the most recent scan data, whereas the *SCANS_HISTORY* holds all scan data.

Name	Type	Description
consignment_id	NUMBER	The consignment identifier reference, as in the *CONSIGNMENTS_HISTORY* table.
item_id	NUMBER	The item identifier reference, as in the *ITEM_HISTORY* table.
scan_type	NUMBER	The type of the scan, see `ScanTypes` enum's ordinal values.
location	VARCHAR	Either the hub identifier, the warehouse's name or the depot identifier, depending on the `scan_type` field's value.
scan_timestamp	DATETIME	The timestamp of the scan.
vehicle_id	VARCHAR	The vehicle's identification, as in the *VEHICLES* table.

Table 49: SCANS table

The table distingueshes between collection and delivery depot scans as well as automatic or manual scans at a hub. Table 50 details the relation between the `scan_type` and `location` fields.

scan_type value	description	location interpretation
0	Scan at the collection depot	Depot identifier.
1	Hub manual scan off the vehicle (unload)	Hub identifier followed by the warehouse's name, i.e., `1 Warehouse A`.
2	Hub manual scan onto the vehicle (load)	Hub identifier followed by the warehouse's name.
3	Hub automatic scan, entering a warehouse	Hub identifier followed by the warehouse's name.
4	Hub automatic scan, leaving a warehouse	Hub identifier followed by the warehouse's name.
5	Scan at the delivery depot	Depot identifier.

Table 50: SCANS table type and location values

5.2.5 Vehicles

5.2.5.1 VEHICLES

The table describes main vehicle properties and their current status at the hub.

Name	Type	Description
vehicle_id	VARCHAR	The vehicle's unique identifier, such as license plate number.
depot	NUMBER	The owner depot of the vehicle.
capacity	NUMBER	The maximum number of items that fit onto the vehicle.
current_job	NUMBER	The current job index as in the *VEHICLE_JOBS* table. *OPTIONAL*.
current_position	NUMBER	The lorry position index if the vehicle is in a warehouse. *OPTIONAL*

Table 51: VEHICLES table

5.2.5.2 VEHICLE_SESSIONS

The table registers the enter and leave time of vehicles into/out of the hub premises.

Name	Type	Description
session_id	VARCHAR	The unique identifier of a hub presence.
vehicle_id	VARCHAR	The vehicle's unique identifier, as in the *VEHICLES* table.
hub	NUMBER	The hub where the vehicle arrived.
arrive_timestamp	DATETIME	The timestamp when the vehicle arrived at the hub.
leave_timestamp	DATETIME	The timestamp when the vehicle left the hub, null if the vehicle is still at the hub.

Table 52: VEHICLE_SESSIONS table

5.2.5.3 VEHICLE_SCANS

The table contains the automatic scan event details for entering and leaving a warehouse.

Name	Type	Description
scan_id	NUMBER	The unique scan identifier.
session_id	VARCHAR	The vehicle's presence session identifier, as in the *VEHICLE_SESSIONS* table.
scan_timestamp	DATETIME	When the scan happened.
warehouse	VARCHAR	The location where the scan happened.
entry	NUMBER	Was the scan on the entry side. `yes = 1`.

Table 53: VEHICLE_SCANS table

5.2.5.4 VEHICLE_ITEMS

The table lists the items detected at each vehicle scan events.

Name	Type	Description
scan_id	NUMBER	The scan identifier, as in the *VEHICLE_SCANS* table.
external_id	VARCHAR	The external item identifier.

Table 54: VEHICLE_ITEMS table

The `external_id` is matched against the `ITEMS.external_id`.

5.2.5.5 VEHICLE_DECLARED

The table lists the items sent from the collection depots. It is used for determining what is on the vehicle when it enters the hub's premises.

Name	Type	Description
vehicle_id	VARCHAR	The vehicle identifier, as in *VEHICLES* table.
declared_timestamp	DATETIME	The declared timestamp.
external_id	VARCHAR	The external item identifier.

Table 55: VEHICLE_DECLARED table

Items belonging to the same declared should contain the same `declared_timestamp` values for all of its items. The `external_id` is matched against the `ITEMS.external_id`.

5.2.6 Learning & Prediction

5.2.6.1 ARX_PREDICTIONS

The table contains the day-by-day predictions, calculated by an Autoregressive model (ARX).

Name	Type	Description
hub	NUMBER	The hub identifier, as in the *HUBS* table.
depot	NUMBER	The depot identifier (as in the *DEPOTS* table), or -1 if the prediction is for the entire hub.
day_run	DATETIME	The timestamp when the prediction was calculated.
day_predict	DATE	The target prediction days.
service_level	NUMBER	The service level, see `ServiceLevel` enum's ordinals.
inbound	NUMBER	The prediction is for the hub-inbound values. `inbound = 1`, `outbound = 0`
unit	NUMBER	The unit of measure, see `UnitStatus` enum's ordinals.
value	DECIMAL	The predicted value.
actual	DECIMAL	The actual value. Null until known.

Table 56: ARX_PREDICTIONS

The actual value is filled in after each day passes.

5.2.6.2 ML_PREDICTIONS

Table contains the during-day predictions calculated by a linear regression model.

Name	Type	Description
hub	NUMBER	The hub identifier, as in the *HUBS* table.
depot	NUMBER	The depot identifier (as in the *DEPOTS* table), or -1 if the prediction is for the entire hub.
day	DATE	The prediction day.
time_of_day	TIME	The prediction time of day.
day_offset	NUMBER	The relative number of days where the prediction points, e.g., `today = 0`, `tomorrow = 1`.
service_level	NUMBER	The service level, see `ServiceLevel` enum's ordinals.
inbound	NUMBER	The prediction is for the hub-inbound values. `inbound = 1`, `outbound = 0`
unit	NUMBER	The unit of measure, see `UnitStatus` enum's ordinals.
current	DECIMAL	The current value at the given day and time.
remaining	DECIMAL	The predicted remaining value.
actual	DECIMAL	The actual value. Null until known.

Table 57: ML_PREDICTIONS table

5.2.6.3 ML_MODELS

The table contains the detailed models on a per depot basis.

Name	Type	Description
hub	NUMBER	The hub identifier, as in the *HUBS* table.
depot	NUMBER	The depot identifier (as in the *DEPOTS* table), or -1 if the prediction is for the entire hub.
day	DATE	The prediction day.
model	CLOB	The XML describing all kinds of models.

Table 58: ML_MODELS table

See 5.6.2.9 for the model XML format.

5.2.7 Scheduling

5.2.7.1 VEHICLE_JOBS

The table contains the vehicle jobs during a vehicle session.

Name	Type	Description
session_id	VARCHAR	The vehicle's session identifier, as in the *VEHICLE_SESSIONS* table.
job_index	NUMBER	The job index.
is_load	NUMBER	Indicate a loading job. `load = 1`
job_start	DATETIME	The start of the job.
job_end	DATETIME	The end of the job.
warehouse	VARCHAR	The warehouse where the vehicle should go, as in the *WAREHOUSES* table.
lorry_position	NUMBER	The lorry position's index, as in the *LORRY_POSITIONS* table.

Table 59: VEHICLE_JOBS table

5.2.7.2 VEHICLE_SLOT_TIMES

The table contains the time of day when vehicles from a depot may enter the hub.

Name	Type	Description
hub	NUMBER	The hub identifier.
depot	NUMBER	The depot identifier.
slot_time_start	TIME	The slot's start time of day.
slot_time_end	TIME	The slot's end time of day.

Table 60: VEHICLE_SLOT_TIMES table

5.2.8 Galassify export tables

5.2.8.1 GAS_DAY_DEPOT_VEHICLES

The table contains aggregated statistics about vehicle contents when they enter or leave the hub's premises used by the **Galassify** application.

Name	Type	Description
session_id	NUMBER	The vehicle session identifier, as in *VEHICLE_SESSIONS* table.
is_enter	NUMBER	The record is for the entered contents (= 1) or the leave contents (= 0).
service_level	NUMBER	The service level, see `ServiceLevel` enum's ordinals.
unit	NUMBER	The unit of measure, see `UnitStatus` enum's ordinals.
value	DECIMAL	The aggregated value.

Table 61: GAS_DAY_DEPOT_VEHICLES table

The vehicle id, enter/leave timestamp and owner depot can be found in the *VEHICLE_SESSIONS* table.

5.2.8.2 GAS_DAY_ITEM_TOTALS

The table contains the end-of-shift inbound/outbound item quantities and the number items left at the hub.

Name	Type	Description
hub	NUMBER	The hub identifier.
depot	NUMBER	The depot identifier.
day	DATE	The day.
service_level	NUMBER	The service level, see `ServiceLevel` enum's ordinals.
unit	NUMBER	The unit of measure, see `UnitStatus` enum's ordinals.
sent_to_hub	DECIMAL	The amount of items sent towards the hub by the depot, i.e., where the `collection_depot = this depot`.
sent_to_hub_others	DECIMAL	The amount of items sent towards the hub by other depots, i.e., where the `delivery_depot = this depot`.
left_at_hub	DECIMAL	The amount of items left at the hub by the depot, i.e., where the `delivery_depot = this depot`.
left_at_hub_others	DECIMAL	The amount of items left at the hub by others, i.e., where the `collection_depot = this depot`.

Table 62: GAS_DAY_ITEM_TOTALS table

5.2.8.3 GAS_DURING_DAY_PREDICTIONS

The table contains the daily predictions in an expanded format with additional details about the prediction model.

Name	Type	Description
hub	VARCHAR	The hub's name.
depot	VARCHAR	The depot's name.
day	DATE	The day of the prediction.
time_of_day	TIME	The time of day of the prediction.
is_inbound	NUMBER	Indicates if this is an hub-inbound prediction (= 1).
service_level	VARCHAR	The service level in textual form.
unit	VARCHAR	The unit of measure in textual form.
day_offset	NUMBER	Which day the prediction is made for, i.e., today = 0, tomorrow = 1, etc.
current	DECIMAL	The current value at the specified time.
remaining	DECIMAL	The predicted remaining value.
actual	DECIMAL	The actual end-day value, null until known.
attribute_values	CLOB	An XML containing the model weights, input attributes and model performance measures.

Table 63: GAS_DURING_DAY_PREDICTIONS table

The depot and service_level may contain ALL indicating a all-depot and all-service-level type prediction.

The attribute_values field contains an XML with the following structure.

```
<attributes mse='double' mae='double' n='int'>
  <{attribute name} [w='double']>double</{attribute name}>
  ...
</attributes>
```

The root node the model's mean square error (mse), the mean absolute error (mae) and the number of input days used during the learning process (n). Child nodes under the root are named according to a simple naming scheme, see section 5.6.2.2 for more details. These child nodes might contain a weight attribute (w) which indicates that the model uses the content value when calculating the prediction. For convenience, the child nodes have the same order as they were selected during the attribute selection progress.

5.2.9 Miscellaneous

5.2.9.1 USERS

Contains the user details, including passwords, email addresses and the associations with hubs and/or depots.

Name	Type	Description
name	VARCHAR	The user's unique name, used for login.
hub	NUMBER	The hub identifier.
depot	NUMBER	The depot identifier, null indicates a hub-level user.
admin	NUMBER	Indicates if the user has administrative rights (= 1).
default_view	NUMBER	Describes which screen to show after logging in. hub overview = 0, warehouse view = 1
password	VARCHAR	The encrypted password.

Table 64: USERS table

If the admin = 1, the user can create, modify and delete other users.

If the depot field is non-null, the ADVANCE Live Reporter restricts the webpages to show only that particular depot's details. Such users can't view the warehouse screens and can't access the user management features even if admin = 1.

The password field contains the BCrypt encrypted password (see http://www.mindrot.org/projects/jBCrypt/). The BCrypt utility class uses the *Blowfish* hashing scheme with adjustable round number to build a hashed password. The usage is straightforward:

```
String hashedPassword = BCrypt.hashpw(plaintextPassword,
  BCrypt.getsalt());
```

To verify a plaintext password, use

```
BCrypt.checkpw(plaintextPassword, hashedPassword);
```

Note that the hashedPassword is not reversible, therefore, it is not suitable for storing passwords that need to be used for login in other systems and services, i.e., logging into a database or webpage.

5.2.9.2 HOLIDAYS

Table contains dates of holidays and other non-working days.

Name	Type	Description
holiday	DATE	The holiday (non-working) date.

Table 65: HOLIDAYS table

The dates are used during the learning process and when retrieveing data for working days only.

5.2.9.3 HUB_DIAGRAM_SCALES

The table lets users specify the so-called **business as usual scale** for the hub diagrams.

Name	Type	Description
name	VARCHAR	The user's name, as in the *USERS* table.
hub	NUMBER	The hub identifier.
unit	NUMBER	The unit-of-measure, see `UnitStatus` enum's ordinals.
value	DECIMAL	The business as usual scale value.

Table 66: HUB_DIAGRAM_SCALES table

5.2.9.4 DEPOT_DIAGRAM_SCALES

The table lets users specify the so-called **business as usual scale** for the depot diagrams.

Name	Type	Description
name	VARCHAR	The user's name, as in the *USERS* table.
hub	NUMBER	The hub identifier.
depot	NUMBER	The depot identifier.
unit	NUMBER	The unit-of-measure, see `UnitStatus` enum's ordinals.
value	DECIMAL	The business as usual scale value.

Table 67: DEPOT_DIAGRAM_SCALES

5.3 Project organization

The ADVANCE Live Reporter is a regular Eclipse J2EE project.

5.3.1 Project settings and requirements

5.3.1.1 Requirements

- 64-bit Graphical Operating System.
- 2+ core, 2GHz CPU.
- 4+ GB RAM.
- 6 GB disk space.
- Java 7 JDK or compatible development environment.
- Eclipse 4.3 or newer (32-bit version recommended).
- SVN client plugin

The project is a regular Eclipse web project. The developers should download the latest Eclipse Java EE developer package from

`http://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplerr`

as adding web capabilities to a non-Java-EE installment is usually error riddled.

After setting up Eclipse and checking out the `advance-live-reporter` project from SourceForge, the compiler and library settings might need to be adjusted:

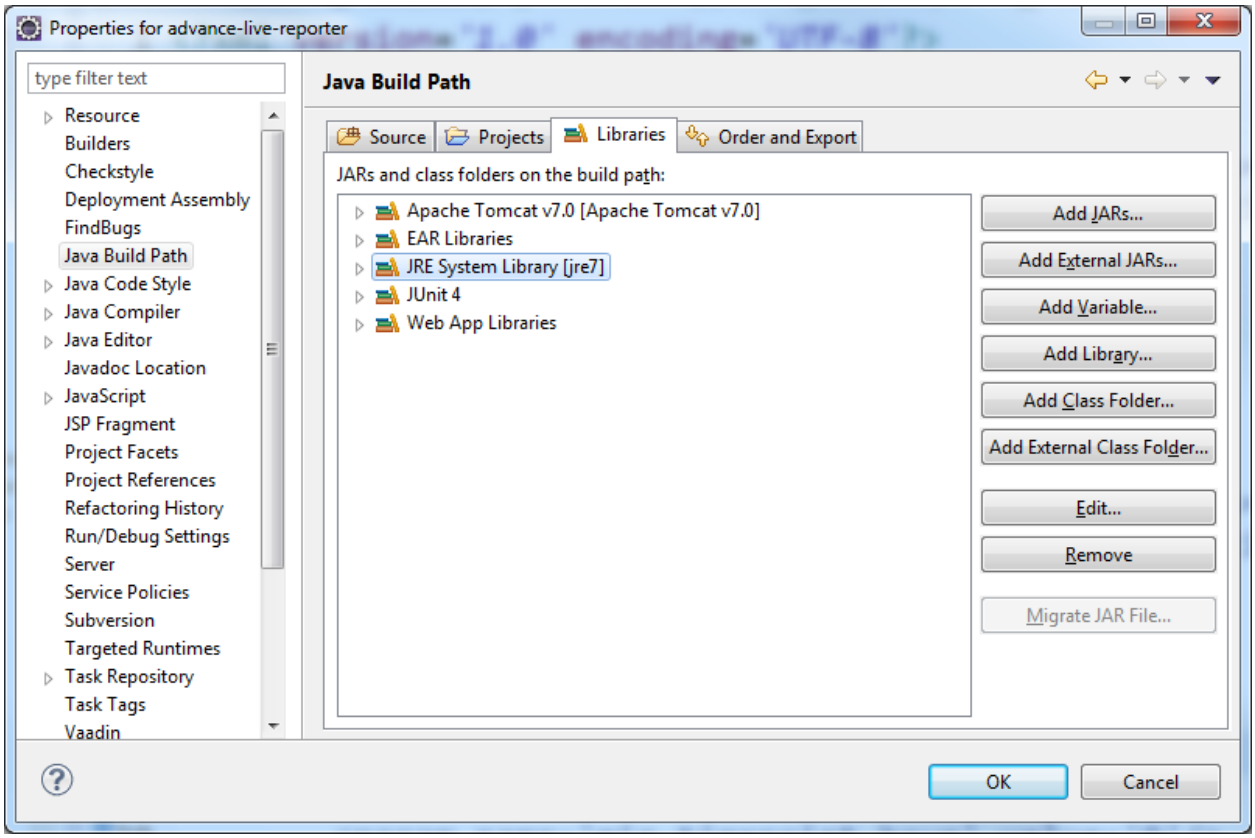

Figure 26: Project setup: change library

Right click on the project in the package explorer, select `Properties`, then select `Java Build Path` and select the `Libraries` page. Adjust the `JRE System Library` entry with the `Edit...` button.

In addition, the compiler needs to be set to **Java 1.7** level. This can be achieved through the *Java Compiler* in the project properties dialog:

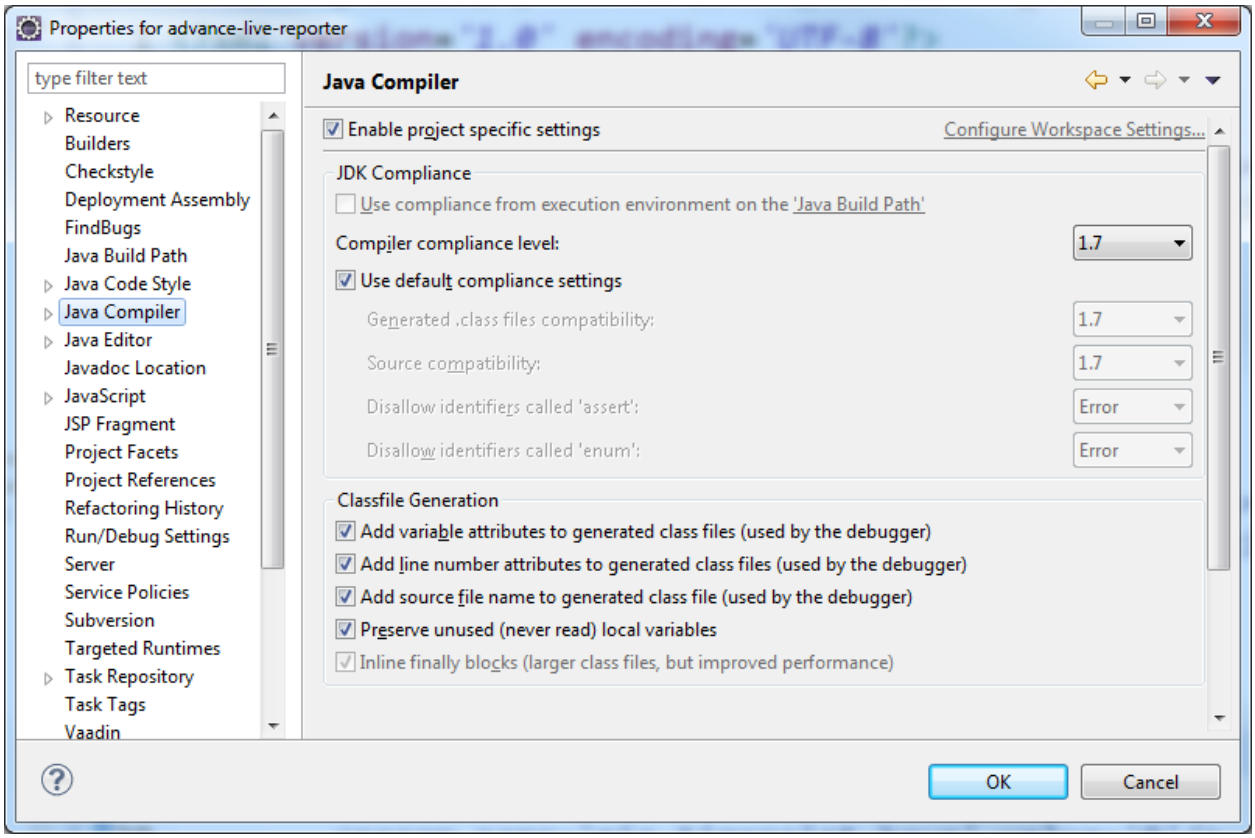

Figure 27: Project setup: change compiler

Most of the time, the *Problems* Eclipse view still contains errors regarding the project facelet incorrectly set up. To fix it, select the error lines, then hit `CTRL+1` for a quick fix, and select *Setup project to use Java 1.7 facelet* or equivalent.

5.3.2 Coding style and settings

The ALR project used the *Checkstyle* plugin to ensure all source code files conform to the preset coding style. In Eclipse, the *Eclipse-CS* plugin (`http://eclipse-cs.sourceforge.net/`) is used.

The style configuration can be found in the `checkstyle.xml` in the projects root directory.

The style settings are numerous, but a few style requirements are listed below.

- Classes, methods, fields must have JavaDoc comments, regardless of their visibility.
- The so-called CamelCase naming convention is used. Classes should start with an uppercase letter; methods and fields with a lowercase letter.

- Blocks begin on the same line as the previous keyword/expression, and must be preceded by a single space.
- Statements such as `if`, `while`, `switch`, etc., need to be followed by a space.

5.3.3 Directory structure

Java source files are located under `/src` directory. Test files are located under `/test` direcory. The web application content is located under `/WebContent`.

The `/WebContent` contains Cascading StyleSheet (CSS) files, JavaScript files, the Java Server Pages (JSP) files and the web application description files. Table 68 contains a brief breakdown of the directories under `/WebContent`.

Directory	Description
`css`	Page header stylesheets.
`css/charts`	The stylesheets of the charts.
`css/datetime`	The jQuery DateTimePicker plugin stylesheets.
`css/dialog`	General jQueryUI dialog stylesheet.
`css/jqtransform`	jQuery styling plugin stylesheet.
`css/tipsy`	The jQuery Facebook-like tooltip stylesheet.
`images`	The page icon and loading animation.
`images/hubdepot`	The images of the hub view.
`images/warehouse`	The images of the warehouse view.
`js/charts`	JavaScript files that create the charts on many screens.
`js/datetime`	The jQuery DateTimePicker and Slider plugin.
`js/dialog`	The Business as usual and Warehouse layout editor dialog JavaScripts.
`js/lib`	The jQuery library.
`js/pages`	The JavaScript files that manage each screen.
`js/tipsy`	The jQuery Facebook-like tooltip JavaScript files.

Table 68: Description of the contents of the ALR WebContent directory.

5.3.4 Package structure

The ADVANCE Live reporter consists of several Java packages, organized by function.

(Since the package names are relatively long, table 69 contains the abbreviation of `alr`, representing the package name of `eu.advance.logistics.live.reporter`.)

Package	Description
`alr.charts`	Data and management classes for the charts.
`alr.db`	Database access classes.
`alr.importdata`	The import webservice class and interfaces.
`alr.importdata.dto`	The record classes supporting the import webservice.
`alr.model`	The record classes based on the database tables and queries.
`alr.prediction.arx`	The blocks for calculating the day-by-day predictions.
`alr.prediction.ml`	The blocks and support classes for learning and calculating the during-day predictions.
`alr.scheduler`	The blocks and support classes to run the scheduler.
`alr.server`	The ADVANCE Flow Engine servlet and various background task classes.
`alr.util`	A few basic utility classes.

Table 69: Packages of ALR

The ALR features hundreds of class files. The following subsections describe some of the most relevant ones.

5.3.5 Notable enumerations

`ServiceLevel`

Enumeration for the supported item service levels:

- STANDARD
- PRIORITY
- SPECIAL
- ALL: indicates a service-level independent evaluation, i.e., combining the values of any service levels.

`UnitStatus`

Enumeration for the unit-of-measure

- VOLUME: indicates the sum of item volumes: `width * height * length`.
- FLOORSPACE: indicates the sum of item floor space: `width * length`.
- PRICE_UNITS: indicates the units after pricing takes place.
- ITEM_COUNT: number of individual items.

`ItemEventTypes`

Enumeration for basic item events relevant for prediction and/or state tracking:

- CREATED: appears in IT system for the first time.
- DECLARED: confirmed delivery.
- SOURCE_SCAN: scanned onto vehicle at the source depot.
- HUB_ENTER: item entered the hub on a vehicle.
- WAREHOUSE_ENTER: item entered a warehouse.
- WAREHOUSE_LOAD: item loaded onto a vehicle in a warehouse.
- WAREHOUSE_UNLOAD: item unloaded from a vehicle in a warehouse.
- WAREHOUSE_LEAVE: item left a warehouse.
- HUB_LEAVE: item left the hub on a vehicle.
- DESTINATION_SCAN: item scanned off a vehicle at the destination depot.

5.3.6 The ALR's web.xml

Every J2EE webapplication contains a `web.xml` file describing the main properties of the application. In ALR, this file is located under `/WebContent/WEB-INF/web.xml`. The schema definition can be found at `http://java.sun.com/xml/ns/javaee`.

The file describes a single welcome file for the application (welcome files are those pages which get shown when the URL doesn't contain any specific page).

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

In order to support the simple web services, the ALR uses JAX-WS (`http://jax-ws.java.net/`) which requires a listener to automatically bind web service calls and marshall/unmarshall web service parameters.

```
<listener>
  <listener-class>
    com.sun.xml.ws.transport.http.servlet.WSServletContextListener
  </listener-class>
</listener>
```

The management interface of the ADVANCE Flow Engine requires an auto-started servlet to function. The servlet responds to the HTTP(S) request of the regular Flow Engine API.

By default, the Flow Engine expects files in the web application directory `WEB-INF/advance-flow-engine`. This can be overridden by specifying a `WorkDir` parameter in the servlet definition. See section 5.5.2 for more details.

```
<servlet>
  <servlet-name>AdvanceFlowEngine</servlet>
    <servlet-class>[...]server.AdvanceFlowEngineServlet</servlet-class>
    <init-param>
      <param-name>WorkDir</param-name>
      <param-value>/home/advance/alr/engine</param-value>
    </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
```

The **import** web service requests need to go through the JAX-WS servlet. See section 5.5.1 for further details. Defining the servlet by its own is not enough, a separate `sun-jaxws.xml` descriptor file is required to map the actual Java class to the service endpoint. See 5.3.7 for details.

```
<servlet>
  <servlet-name>importdata</servlet-name>
    <servlet-class>
      com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The ALR supports background tasks which can be started based on timing information, on minute accuracy. In order to start tasks, a so-called Crontab servlet is running in the background, which periodically checks if tasks have met their starting (time) conditions. See section 5.5.3 for more

details.

```
<servlet>
  <servlet-name>Crontab</servlet-name>
  <servlet-class>[...]server.CrontabServlet</servlet-class>
  <load-on-startup>3</load-on-startup>
</servlet>
```

Servlets that need to be accessible through an URL require servlet mappings.

```
<servlet-mapping>
  <servlet-name>AdvanceFlowEngine</servlet-name>
  <url-patern>/AdvanceFlowEngine</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>importdata</servlet-name>
  <url-pattern>/importdata</servlet-name>
</servlet-mapping>
```

In case the ALR and all of its pages need to be secured behind HTTPS inside the web container, a security constraint need to be added.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>securedapp</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

To temporarily disable the security mechanism, the `url-pattern` can be changed something else which doesn't match the default JSP pages, e.g., `<url-pattern>/xyz*</url-pattern>`.

5.3.7 The sun-jaxws.xml

The default JAX-WS requires a mapping file to match the web service endpoint with actual Java classes. These endpoint classes can be setup in the `/WebContent/WEB-INF/sun-jaxws.xml` file.

Currently, only the **import** web service has mapping defined.

```
<endpoint
  name='importdata'
  implementation='[...]importdata.ImportSEI'
  url-pattern='/importdata'
>
```

5.3.8 Database connection configuration

Java's JDBC library is wrapped by the DB class to provide convenience operations while communicating with databases, including *varags* query parameters, callback style resultsets, etc.

The DB class expects a /db.xml file to be present in the classpath root. The file describes a set of database connection parameters. The db.xsd that describes the XML format can be found inside the akarnokd-tools.jar library. A simple breakdown of the structure is as follows.

The root databases element contains any number of database child elements.

```
<database>
  <database id='default'>
    ...
  </database>
  ⋮
</database>
```

The parameterless connect() static method will look for the database entry where the id attribute is default (overload is available for specifying any other id).

```
try (DB db = DB.connect("other-db")) {
}
```

Inside each database element, the connection information is present.

```
<database id='default'>
  <driver-class>com.mysql.jdbc.Driver</driver-class>
  <connection-url>jdbc:mysql://localhost/advance_db</connection-url>
  <user>advance</user>
  <password encoded='false'>password</password>
  <schema>advance_db</schema>
  <max-connection>10</max-connection>
</database>
```

The child elements describe the regular driver and connection parameters of a database. If the password/@encoded attribute is true, the element's content is expected to be in Base64.

The uniqueness of the `id` parameter values are not enforced. When the configuration is parsed and multiple entries with the same id are found, the last one wins.

5.3.9 Email sender configuration

The ALR gives the opportunity to send alerts and notifications to users via Email. The `EmailSender` class helps sending such messages by properly paramterizing the underlying *JavaMail* library.

In order to send an email, an SMTP(s) server is required. The `/email.xml` file in the classpath allows the configuration of accessing such SMTP(s) server. The schema for this XML can be found in `/email.xsd`.

The root `email` element may contain many `connection` child elements. Each child element defines an `id` attribute. The parameterless `EmailSender` constructor expects a connection with `id='default'` to be present (alternative constructors are available to select a particular connection).

```
<email>
  <connection id='default'>
    ...
  </connection>
  ⋮
</email>
```

Each `connection` element has several mandatory child elements.

```
<connection id='default'>
  <protocol>smtps</protocol>
  <host>smtp.server.org</host>
  [<port>888</port>]
  <auth>true</auth>
  <sender>default-sender@server.org</sender>
  <subject>default subject</subject>
  [<user>user</user>
  <password encoded='false'>password</password>]
</connection>
```

The underlying JavaMail library supports `SMTP` and `SMTPS` protocols (lowercase in the XML). The `port` element is optional, if missing, the protocol default ports are used. If the `auth` element has `true`, the `user` and `password` elements need to be present. The `sender` is set as the default email sender (i.e., it will appear in the *from:* field). If the `password/@encoded` attribute is true, the element's content is expected to be in Base64.

The uniqueness of the `id` parameter values are not enforced. When the configuration is parsed and multiple entries with the same id are found, the last one wins.

5.3.10 Logging configuration

The ALR uses the *log4j* library for logging. By default, the library expects a `log4j.xml` (or `log4j.properties`) file to be present in the classpath root. ALR has the XML-based configuration. Since most environment can't go onto the internet to download the log4j DTD, the appropriate `/log4j.dtd` is available in the classpath as well.

Configuration of the log4j can be quite complicated, please refer to the

http://wiki.apache.org/logging-log4j/Log4jXmlFormat

(only available from archive) wiki page.

The project-default XML sets up the DEBUG or above level events to be printed to the standard output, and the ERROR or above level events to be printed to the standard error.

A logger can be requested from a logging factory:

```
Logger LOGGER = LoggerFactory.getLogger(CurrentClass.class);
```

The ALR the Simple Logging Facade (slf4j) which wraps and all common logging frameworks to Log4J logger:

- java.util.logging calls
- Commons Logging calls

Some libraries use one or the other. The wrapping ensures that all logging activity goes into the same destination logger.

5.3.11 Libraries

The ALR uses several open-source libraries to support the operation. Table 70 lists the libraries and their brief descriptions.

Name	Ver.	License	Description
`advance-flow-engine`	1.0	Apache 2.0	The ADVANCE Flow Engine library
`akarnokd-tools`	-	Apache 2.0	General tools library from the developers of ALR. `https://github.com/akarnokd/akarnokd-tools-and-utils`

Table 70: ALR libraries

Name	Ver.	License	Description
annotations	-	New BSD	JSR 305 Annotations. https://code.google.com/p/jsr-305
commons-beanutils	1.8	Apache 2.0	JavaBeans utility library. http://commons.apache.org/proper/commons-beanutils/
commons-codec	1.6	Apache 2.0	Encoding and decoding library. http://commons.apache.org/proper/commons-codec/
commons-collections	3.2.1	Apache 2.0	Java Collections extension. http://commons.apache.org/proper/commons-collections/
commons-lang	2.6	Apache 2.0	Java language extension. http://commons.apache.org/proper/commons-lang/
commons-logging	1.1.1	Apache 2.0	Logging library. http://commons.apache.org/proper/commons-logging/
commons-math3	3.2	Apache 2.0	Mathematical library. http://commons.apache.org/proper/commons-math/
ezmorph	1.0.6	Apache 2.0	Object transformation library. http://ezmorph.sourceforge.net/
guava	14.0.1	Apache 2.0	Google collections library. https://code.google.com/p/guava-libraries/
javamail	1.4	CDDL/GPLv2	Email sender library. http://www.oracle.com/technetwork/java/javamail/index.html
jaxws-rt	2.2.8	CDDL/GPLv2	JAX-WS reference implementation. http://jax-ws.java.net/
joda-time	2.2	Apache 2.0	Date and time library. http://joda-time.sourceforge.net/
json-lib	2.4	Apache 2.0	JSON conversion library. http://json-lib.sourceforge.net/
log4j	1.2.17	Apache 2.0	Logging library. http://logging.apache.org/log4j/1.2/
mysql-connector	5.1.25	GPLv2	MySQL database driver. http://dev.mysql.com/downloads/connector/j/

Table 70: ALR libraries

Name	Ver.	License	Description
reactive4java	0.97	Apache 2.0	The reactive programming library. https://code.google.com/p/reactive4java/
slf4j	1.6.1	Apache 2.0	Simple logging facade to wrap all kinds of logging frameworks. http://www.slf4j.org/
trove	3.0.3	LGPL	Primitive collections library. http://trove.starlight-systems.com/

Table 70: ALR libraries

5.4 Screen files

The ADVANCE Live Reporter is a regular Java 2 Enterprise Edition webapplication and uses Java Server Pages (JSP) and employs Asynchronous JavaScript (AJAX) technology extensively.

The files can be grouped into three categories:

- header files (*.jspf)
- screen pages (*.jsp)
- asynchronous endpoint files (api-*.jsp)

The screen pages and the asynchronous endpoint communicates via JavaScript Object Notation (JSON) objects.

All JSP files, except the index.jsp and login.jsp, verify if the user is logged in and is allowed to view the given screen. The files are located under the project's /WebContent directory.

Table 71 briefly describes each of the relevant files.

File	Description
api-baus.jsp	Saves the so-called business as usual scale values for the current hub or depot.
api-daybyday.jsp	Returns the chart data for showing the day-by-day prediction diagrams for the current hub or depot.
api-duringday.jsp	Returns the chart data for showing the during-day prediction diagrams for the current hub or depot.
api-layout.jsp	Returns and saves the layout of the storage areas within a warehouse.

Table 71: JSP files of ARL

File	Description
api-warehouse-l1.jsp	Returns the chart data for the warehouse overview screen (warehouse level 1).
api-warehouse-l2.jsp	Returns the chart data for the individual warehouse content screen (warehouse level 2).
api-warehouse-l3.jsp	Returns the chart data for the detailed warehouse content screen (warehouse level 3).
api-summary.jsp	Returns the chart data for the hub or depot overview screen.
api-users.jsp	Returns and saves the user settings for an administrator.
daybyday.jsp	The day-by-day screen.
duringday.jsp	The during-day screen.
header.jspf	The header and menus displayed on each non-warehouse screens.
index.jsp	The main entry point of the application, contains the username/-password form.
login.jsp	A non-visible page that performs the user authentication and selects the first page to display.
logout.jsp	A non-visible page that terminates the user's session and returns him/her to the index page.
warehouse-header.jspf	The warehouse header file showing the main controls for each warehouse screen.
warehouse-l1.jsp	The warehouse overview (level 1) screen.
warehouse-l2.jsp	The warehouse comparison (level 2) screen.
warehouse-l3.jsp	The warehouse detailed (level 3) screen.
summary.jsp	The hub-level or depot level summary screen.

Table 71: JSP files of ARL

5.4.1 General pages overview

In the application, every html page is combined from two parts: from the main page and from the header page, which is included into the main page at the server run time.

There are number of two types of header page: the non-warehouse header and the warehouse-header. Both header contains link buttons, html elements for user's settings and a page-embedded JavaScript (header-script) which is written in jQuery plug-in style.

The main page contains the html skeleton and that div tag into where the chart is generated dynamically. The main page has a page-script which is stored in a separate JavaScript file and

written in jQuery plug-in style. This page-script responsible for calling the init function of the header-script and handling the generated change event by header-script to refresh the chart.

The table 72 summarizes the non-warehouse header and the main pages of it:

Header page	**Header file**	**Hader script**
Non-warehouse header	WebContent/header.jspf	Embedded into header.jspf
textbfMain page	**Main page file**	**Page script file**
Summary	WebContent/summary.jsp	WebContent/js/pages/sum-page.js
Next days	WebContent/daybyday.jsp	WebContent/js/pages/dayby-page.js
During day	WebContent/duringday.jsp	WebContent/js/pages/duringday-page.js

Table 72: Non-warehouse main pages, headers and script files

.

The table 73 summarizes the warehouse header and the main pages of it:

Header page	**Header file**	**Hader script**
Warehouse header	WebContent/warehouse-header.jspf	Embedded into the warehouse-header.jspf
textbfMain page	**Main page file**	**Page script file**
Warehouse Level 1	WebContent/warehouse-l1.jsp	WebContent/js/pages/warehouse-l1-page.js
Warehouse Level 2	WebContent/ warehouse-l2.jsp	WebContent/js/pages/warehouse-l2-page.js
Warehouse Level 3	WebContent/ warehouse-l3.jsp	WebContent/js/pages/warehouse-l3-page.js

Table 73: Non-warehouse main pages, headers and script files

.

5.4.2 Posting user's settings to a requested page

In order to open the same page in more than one browser tabs irrespectively to each other, the application does not use session for storing the user's settings. That is the reason, why the actual values of them are stored in the form hidden variables in order to post them to the required main page when the user clicks onto any menu link. (Noted that the application has a "USER" session, but this does not store the user's display settings.)

The lifecycle from post the hidden variables to initialize the requested ***non-warehouse*** *page (For example, the requested main page is* `summary.jsp`*):*

- After clicking onto the menu (ex.: "Summary"), the menu click event is handled by the header-script in order to:

  - refresh the values of the hidden variables in the form
  - submit the form to the new requested main page

- The server includes the `header.jspf` file into the `summary.jsp`, then executes the jsp scriptlet and expressions of the `header.jspf` file, such as:
  - The `HubDepotSwitch` class is initialized with the values of the posted hidden variables or the default values (if hidden variables are undefined).
  - In the header-script, the JavaScript object called "active" stores the posted values of the hidden variables for initializing the html elements. The right side of this JavaScript object is set by evaluating the jsp expressions.
- The compiled `summary.jsp` page is downloaded and the `$.summarypage("init")` calls the `init : function()` public function of the summary page-script to run the complex initialization:
  - The `$.headoption("init", "summary")` calls the header-script function named `init : function("summary")` to initialize the "active" object firstly; set the html elements based on this "active" object secondly; start listeners for user events thirdly.
  - Page-script listeners are started to catch the change event handlers in order to refresh the chart.

The table 74 shows the relationship between the non-warehouse header html element of the user's settings, the header-script "active" object and the posted hidden variables of the header form.

User's settings	Type part (hub or depot) of the value of the selected store
HTML tag	$("div#design-selector span#active-select") generated from $("select#store_selector")
Field of active object	active.store.type
Hidden variable	$('#hd_form input[name="store_type"]')
User's settings	Id part of the value of the selected store
HTML tag	$("div#design-selector span#active-select") generated from $("select#store_selector")
Field of active object	active.store.id
Hidden variable	$('#hd_form input[name="store_type"]')
User's settings	Unit status: Punts, Items, Fspc
HTML tag	$('div[id^="unit"]')
Field of active object	active.unit
Hidden variable	$('#hd_form input[name="unit"]')
User's settings	DateTime calendar or Datetime minus and plus
HTML tag	$("input#date_time")
Field of active object	active.date
Hidden variable	$('#hd_form input[name="datetime"]')
User's settings	Orient (Inbd, Outbd) only for during day page
HTML tag	$('div[id^="orient"]')
Field of active object	active.dd_orient
Hidden variable	$('#hd_form input[name="dd_orient"]')
User's settings	Snip (Eco, Prem, NGen) only for during day page
HTML tag	*The header does not contain html tag for it*
Field of active object	active.dd_snips
Hidden variable	$('#hd_form input[name="dd_snips"]')
User's settings	Actual cursor settings only for during day page
HTML tag	The header does not contain html tag for it
Field of active object	active.dd_cursor
Hidden variable	$('#hd_form input[name="dd_cursor"]')

Table 74: Relationship between non-warehouse HTML element and the user's settings

The lifecycle from post the hidden variables to initialize the requested ***warehouse*** *page (For*

example, the requested main page is `warehouse-l2.jsp`):

- After clicking onto the warehouse-level button, the click event is handled by the header-script in order to:
  - refresh the values of the hidden variables in the form
  - submit the form to the new requested main page
- The server includes the `warehouse-header.jspf` file into the `warehouse-l2.jsp`, then executes the jsp scriptlet and expressions of the `warehouse-header.jspf` file, such as:
  - The `WarehouseSwitch` class is initialized with the values of the posted hidden variables or the default values (if hidden variables are undefined).
  - In the header-script, the JavaScript object called "active" stores the posted values of the hidden variables for initializing the html element. The right side of this JavaScript object is set by evaluating the jsp expressions.
- The compiled `warehouse-l2.jsp` page is downloaded and the `$.warehousel2page("init")` calls the `init : function()` function of the warehouse-l2 page-script to run the complex initialization:
  - The `$.warehousemenu("init", "l2")` calls the header-script function named "init : function("l2")" to initialize the "active" object firstly; set the html elements and event listeners based on the unique Id ("l2") of the requested page secondly.
  - Page-script listeners are started to catch the change event handlers in order to refresh the chart.

The table 75 shows the relationship between the warehouse header html element of the user's settings, the header-script "active" object and the posted hidden variables of the header form.

User's settings	Warehouse part of the warehouse selector
HTML tag	$('td[id^="warehouse_"]')
Field of active object	active.warehouse
Hidden variable	$('#warehouse_form input[name="warehouse_name"]')
User's settings	Option part of the warehouse selector
HTML tag	$('td[id^="option_"]')
Field of active object	active.option
Hidden variable	$('#warehouse_form input[name="warehouse_option"]')
User's settings	Order part of the warehouse selector
HTML tag	$('td[id^="order_"]')
Field of active object	active.order
Hidden variable	$('#warehouse_form input[name="storage_order"]')
User's settings	*Option part of the warehouse selector only for the warehouse-level 3*
HTML tag	*$('td[id^="option_"]')*
Field of active object	*active.l3option*
Hidden variable	*$('#warehouse_form input[name="warehouse_l3_option"]')*

Table 75: Relationship between warehouse HTML element and the user's settings

5.4.3 Adding a new non-warehouse page

As written above, every page has two parts: the main page and a common header page which is included into the main page. This means a new main page has to be created and the header page has to be modified such as add a new menu link and add an event handler for it.

Some rules for creating a new jsp main page:

- The next meta tags should add to the page:

```
<meta name="viewport" content="width=device-width,
  initial-scale=1.0, maximum-scale=1.0, user-scalable=no">
<meta http-equiv="Content-Type" content="text/html;
  charset=ISO-8859-1">
<meta http-equiv="Cache-Control" content="no-cache,
  no-store, must-revalidate">
<meta http-equiv="Pragma" content="no-cache">
```

- The next css styles have to include for showing the header properly:

```
<link rel="stylesheet" type='text/css'
  href="css/hubdepot-header.css">
<link rel="stylesheet" type='text/css'
  href="css/charts/common.css">
<link rel="stylesheet" type='text/css'
  href="css/dialog/sle.css">

<link rel="stylesheet" type="text/css"
  href="css/datetime/jquery-ui.css">
<link rel="stylesheet" type="text/css"
  href="css/datetime/jquery-ui-timepicker-addon.css" >
<link rel="stylesheet" type="text/css"
  href="css/jqtransform/jqtransform.css" >
```

- The next jQuery plug-in and JavaScript libraries have to include for working the header properly:

```
<script src="js/lib/jquery-1.8.1.js" type="text/javascript">
</script>
<script src="js/lib/jquery-ui-1.10.3.js" type="text/javascript">
</script>

<script src="js/dialog/baus.js" type="text/javascript">
</script>
<script src="js/dialog/sle.js" type="text/javascript">
</script>

<script src="js/datetime/jquery-ui-timepicker-addon.js"
  type="text/javascript"></script>
<script src="js/datetime/jquery-ui-sliderAccess.js"
  type="text/javascript"></script>
<script src="js/lib/moment.js" type="text/javascript"></script>
<script src="js/lib/jquery.jqtransform.js"
  type="text/javascript"></script>
```

  Noted that the jQuery Core and the jQuery UI have to be included firstly to the page.

- Include the "header.jspf" header file into the body tag of the html:

```
<%@include file='header.jspf' %>
```

- With the same level of the body tag, add a script tag in where you can define the entry point of your separated page-script or you can embed your script directly into this main page. Every main page has a string type unique Id which is sent to the header-script at the initialization. The reason of it is to be able to modify the business logic of the header based on the actual page. The page-script has to begin with the init function of the header-script, such as:

```
<script type="text/javascript">
$.headoption("init", "summary");
</script>
```

  In the example above, the public "init : function(param)" function of the header-script is called with the unique Id of the summary page ("summary").

- If the page-script is stored in a separate file, include the script tag `<script src="..." type="text/javascript"></script>` into the html header-part of this new main page.

After creating the main page, some parts of the "header.jspf" have to be modified:

- A new menu link has to be added to the header. The section of the menu links is found under the `<div class="menu-container-div">` div tag. Noted that every menu has an unique Id with prefixed `m_"` and has the same init style. For example a new "Example Menu" is able to added as: `<div id="m_example_menu"class="switch-box-div switch-box-inactive main-menu">Example Menu</div>`.

- Make relationship between the created page Id and this new menu Id. The header-script init function calls the `setSelectedMenu : function()` private function which sets the selected menu based on the page Id. This unique page Id parameter is stored into the `atPage` JavaScript object, which can be reached in the `setSelectMenu` function. So this function has to be extended with adding a new "else if" statement. The `menuName` is the unique Id of the menu link.

- Add click event listener for this new menu. The header-script init function calls the `menuSelectHandler : function()` private function which handles the menu click event. This function has to be extended with adding the new jQuery `.on()` function with `click` parameter to the new menu. The second parameter of this `.on()` function is an anonymous function which calls the `formSelector : function(actionURL)`, in where the `actionURL` parameter is the file name of the new main page.

5.4.4 Adding a new warehouse page

As the non-warehouse pages too, warehouse pages have two parts: the main page and the header page which is included into the main page.

Some rules for creating a new jsp main page:

- The next meta tags should add to the page:

```
<meta name="viewport" content="width=device-width,
  initial-scale=1.0, maximum-scale=1.0, user-scalable=no">
<meta http-equiv="Content-Type" content="text/html;
  charset=ISO-8859-1">
<meta http-equiv="Cache-Control" content="no-cache,
  no-store, must-revalidate">
<meta http-equiv="Pragma" content="no-cache">
```

- The next css styles have to include for showing the header properly:

```
<link rel="stylesheet" type="text/css"
  href="css/warehouse-header.css">
<link rel="stylesheet" type="text/css"
  href="css/charts/common.css">

<link rel="stylesheet" type="text/css"
  href="css/datetime/jquery-ui.css">
```

- The next jQuery plug-in and JavaScript libraries have to include for working the header properly:

```
<script src="js/lib/jquery-1.8.1.js" type="text/javascript">
</script>
<script src="js/lib/jquery-ui-1.10.3.js" type="text/javascript">
</script>
```

  Noted that the jQuery Core and the jQuery UI have to be included firstly to the page.

- Include the "warehouse-header.jspf" header file into the body tag of the html:

```
<%@include file="warehouse-header.jspf" %>
```

- With the same level of the body tag, add a script tag in where you can define the entry point of your separated page-script or you can embed your script directly into this main page. Every main page has a string type unique Id which is sent to the header-script at the initialization. The reason of it is that the business logic of the header is depend on the actual warehouse level page. The page-script has to begin with the init function of the header-script, such as:

```
<script type="text/javascript">
$.warehousemenu("init", "l1");
</script>
```

In the example above, the public "init : function(param)" function of the header-script is called with the unique Id of the warehouse-level1 page ("l1").

- If the page-script is stored in a separate file, include the script tag `<script src="..." type="text/javascript"></script>` into the html header-part of this new main page.

After creating the main page, the "warehouse-header.jspf" has to be modified:

- A new link has to be added with a unique Id to the header in order to reach this new page. However, based on the business logic, warehouse level pages can be reached only sequentially. If this new page should reach from every warehouse level (i.e. random as at the non-warehouse header), you have to define a place to this new link under the `<div class="warehouse-menu">` div tag, because there is not any predefined place for it over against being it in the non-warehouse header for new menu.
- Modify the init method based on the created page Id in order that which level button and which part of the warehouse selector is shown. This means that the init function has to be extended with adding a new "else if" statement.
- Add click event listener for this new link. In the header-script, the `setUpDown : function(imgCSS, actionURL)` private function handles the level link sequential button. Based on this function, a new one has to be created for the new link and this new function has to be added into the init function. Additionally: the setUpDown function calls the `getAllEnumMenu : function()` private function to gather the user's settings based on the `atPage` parameter which stores the unique Id of the actual warehouse level page. So this `getAllEnumMenu` function has to be extended with adding a new "else if" statement based on the `atPage` parameter and should be called from the new handler function.

5.4.5 Adding a new global dialog

The non-warehouse header contains the global settings icon. After clicking onto it and selecting a menu, the modal dialog window is opened. Every modal dialog has its own dialog-script which is written in jQuery plug-in style and stored in a separated JavaScript file as the table 76 shows it below.

Dialog	Dialog script file
Business as usual scale	WebContent/js/dialog/baus.js
Warehouse Layout Editor	WebContent/js/dialog/wle.js

Table 76: Existing dialog scripts

Create a new menu with modal dialog step by step:

- A new menu link has to be added to the global settings. The section of setting menu links is found under the `<div id="h_set_details" class="settings-details settings-details-hide">` div tag. Noted that every menu link has unique Id and the same css style.
- A new modal dialog window has to be added with `dialog_` prefix unique Id under the `<div class="header-div">`. As the dialog window is managed by the jQuery UI Dialog component, so the style of this div is fixed.
- The `settingsIconHandler : function()` private function handles the event handling of the global settings icon and the menu. This function has to be extended with adding the new jQuery `.on()` function with `click` parameter to the new menu link. The second parameter of this `.on()` function is an anonymous function in where the dialog-script plug-in is called.

  â€¢ After writing the dialog-script file, include the script tag `<script src="..." type="text/javascript"></script>` into the html header-part of every non-warehouse main page.

5.5 ALR services

5.5.1 Import

Since the ALR has its own database with the operational data, it is essential that real-time updates are transferred to it. Therefore, a web service is provided that lets external systems send data to the ALR.

The web service is implemented upon the JAX-WS library and can be found in `eu.advance.logistics.live.reporter.importdata.ImportSEI` class. JAX-WS allows the definition of web services and methods via Java annotations, but requires custom servlet to integrate with any J2EE container. The library automatically generates the WSDL and manages the SOAP message marshalling/unmarshalling. The WSDL can be requested through the usual URL parameter:

```
/advance-live-reporter/importdata?wsdl
```

The ImportSEI has one single method: importData, which has a composite input record and returns a textual status or error code once the import finished.

The input consists of several (optional) tables with simple, flat record structure. The eu.advance.logistics.live.reporter.importdata.dto.DataPack class wraps them.

```
class DataPack {
  String userName;
  String password;
  ImportConsignment[] consignments;
  ImportItem[] items;
  ImportScan[] scans;
  ImportTerritory[] territories;
  ImportVehicleSession[] vehicleSessions;
  ImportVehicleScan[] vehicleScans;
  ImportVehicleItem[] vehicleItems;
  ImportVehicleDeclared[] vehicleDeclared;
}
```

The authentication is performed via this input data as well (instead of using transport level authentication). The USERS table is checked against the provided user and password. The user should have admin = 1 or else the import is rejected.

The array type fields in the DataPack are record classes that mostly map to the tables with the same name directly.

There exists, however, some differences between the table fields and the class fields of the ImportConsignment class. Namely, in the class, the hub, collectionDepot, deliveryDepot, collectionPostcode and deliveryPostcode fields are defined as Strings, which will be mapped to a numeric index based on the HUBS (see Table 39), DEPOTS (see Table 40) and POSTCODES (see Table 41) table. If a particular hub or postcode is not in the respecitve table, a new id is generated and the CONSIGNMENTS table will contain this new value (see Table 43). Tables with _HISTORY partner receive the same content.

The second difference is that each of the record classes have the isDeleted optional field that indicates a particular record needs to be deleted.

5.5.2 Flow Engine Servlet

In order to access the ADVANCE Flow Engine running inside the ALR through the regular, HTTP(S) based API, a servlet is developed which converts the J2EE HttpServletRequest calls into the Flow Engine's AdvanceEngineControl method invocations.

The servlet eu.advance.logistics.live.reporter.server.AdvanceFlowEngineServlet currently supports (and requires) only HTTP Basic authentication. The servlet itself is stored in the servlet context under ADVANCE_FLOW_ENGINE_KEY and static getServlet() convenience methods are provided to retrieve it.

The servlet manages both immediate and streaming response types.

The servlet also hosts the Engine and can be retrieved via getEngine() method call. The call returns a direct access to the Engine, but if function-authorization is required, the getEngine(String) overload returns a CheckedEngineControl for the specified user.

By default, the engine stores its configuration and data files in the /WEB-INF/advance-flow-engine, but can be overridden in the web.xml with an init-param named WorkDir.

5.5.3 Crontab

To support the day-to-day operation of the ALR, background task need to be run at specific times and/or days independent of the user, such as the periodic learning and prediction, database housekeeping and export data preparation.

Such tasks are managed by the CrontabServlet class, the servlet manages a scheduled thread-pool which periodically checks the staring condition of the registered tasks, and executes them. Tasks are regular Java classes which need to implement the CrontabTask interface. The interface has a single textttexecute() method which receives a contextual record of type CrontabTaskSettings. If necessary, the *current time* can be extracted from the CrontabTaskSettings.lastCheck field, which allows the execution of the task on a virtual-time (i.e., not real-time, run for historical days) manner. The parameters are provided through a ParameterHashMap class.

The starting time condition is maintained by the CrontabTime class. Each field (years, months, days, weekdays, hours and minutes) can be null indicating the any-time case for that field, whereas if not null, it lists the exact time-parts that the current time checked by the CrontabServlet should match. For example, if

```
years = null,
months = null,
days = null,
weekdays = 1,
hours = { 6, 12, 18 },
minutes = { 0 }
```

indicates that the task will run on each monday at exactly 6AM, 12PM and 6PM.

The list of tasks and their parameters are listed in the WEB-INF/classes/crontab.xml file,

which has the following structure:

```
<crontab>
  <task id='string' years='ints' months='ints' days='ints'
    weekdays='ints' hours='ints' minutes='ints'>
      <param name='string' value='any'/>
      ⋮
  </task>
  ⋮
</crontab>
```

The root `crontab` element might contain zero or more `task` child elements.

Each `task` contains an unique `id` identifier field. The time pattern can be specified through the `years`, `months`, `days`, `weekdays`, `hours` and `minutes`. Each may contain a single star (`*`) indicating an any-time case, or a list of comma separated values. The `weekdays` specifies the day of week where monday is 1 and sunday is 7.

Under the `task` element, zero or more `param` parameter element might be listed. Such `param` parameters contain a `name` attribute and `value` attribute, both are task-specific and whether or not they are required.

In the default ALR setup, the following tasks have been setup:

- **ARXRunner**: executes the day-by-day learning task.
- **MLLearner**: learns the within-day models on Sundays.
- **MLPredictor**: calculates the predictions real-time as a timepoint passes.
- **MLOvernight**: calculates the end-of-day item values and left-at-hub values before each new shift starts (e.g., 5AM).

5.6 Prediction routines

5.6.1 Day-by-day prediction

The day-by-day prediction calculates via an Autoregressive model with exogenous parameters (ARX), which is basically a linear model.

$$x_t = \sum_{i=1}^{p} (a_i x_{t-i} + \sum_{j=1}^{m} b_j u_{t-i,j}) \tag{1}$$

In the ALR, the ARX model has the following properties:

- Calculates a model for predicting the daily declared item values (x_t). The input is the declared item values on previous workdays (x_i).
- The model order is adjustable (default is p = 10).
- Learn from an adjustable period (default is one year).
- The exogenous parameters are binary variables for the days of week. A parameter is 1 if the day of week of the source data is the same. (m = 5).
- Weekends and holidays are ignored.
- Input data can be normalized (default is yes).
- The split percent between training and test set is adjustable (default is 75% training).

By using the Yule-Walker equations, the ARX problem can be written in a matrix form and can be solved by least-squares methods.

$$Y^T = \begin{bmatrix} x_t & x_{t+1} & \cdots & x_{t+p-1} \end{bmatrix} \tag{2}$$

$$A = \begin{bmatrix} x_{t-1} & x_{t-2} & \cdots & x_{t-p} & u_{1,t-1} & u_{2,t-1} & \cdots & u_{m,t-1} \\ x_t & x_{t-1} & \cdots & x_{t-p+1} & u_{1,t} & u_{2,t} & \cdots & u_{m,t} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{t+p-2} & x_{t+p-3} & \cdots & x_{t-1} & u_{1,t+p-2} & u_{2,t+p-2} & \cdots & u_{m,t+p-2} \end{bmatrix} \tag{3}$$

$$X^T = \begin{bmatrix} a_1 & a_2 & \cdots & a_p & b_1 & b_2 & \cdots & b_m \end{bmatrix} \tag{4}$$

$$Y = AX \tag{5}$$

In general, the matrix equation (5) is solved by the least-squares formula:

$$X = (A^T A)^{-1} A^T Y \tag{6}$$

but usually the $(A^T A)^{-1}$ can't be computed, therefore, a pseudoinverse function such as the Singular Value Decomposition is used. In the ALR the *commons-math* library's `SingularValueDecomposition` class is used.

The learning and prediction is performed on a daily basis for multiple dimensions:

- hub and depots,
- for each service level, plus service level independently,
- for each unit of measure
- for hub-inbound and hub-outbound directions,

The relevant ARXRunnerTask is an optimized class to perform the data retrieval, learning and predicting by the dimensions and saving the results into the database. The model (e.g., the weights) are not saved.

The task supports the following parameters:

Name	Type	Description
hub	long	Limit the consignment data to the particular hub.
days	int	The number of calendar days to learn from.
horizon	int	The number of days to predict, starting from the current day.
shift-start	int	The hour of day when a new shift starts.
use-blacklist	boolean	Ignore holidays and weekdays.
model-size	int	The model size (number of previous days to use).

Table 77: ARXRunnerTask parameters

In the first step, the existing predictions calculated at the current day are deleted via PredictionDB.deleteARXPredictions() method.

Next, the daily declared item values are retrieved from database for the specified number of days. These declared values are available on a per depot level and for each service level, therefore, a secondary aggregation is performed to calculate the hub level and service level-independend daily declared item counts in aggregateForHubSL() method.

The third step is to calculate the model and predictions for all kinds of value dimensions in calculate() method. The method first projects the daily declared data from the previous step to a specific depot, service level, unit of measure, then calls the ARXLearner.learn() method to calculate the model coefficients. The learn(), in turn uses the KMeansARX class – common with the ADVANCE Flow Engine – to perform the actual learning. In order to be able to calculate the coefficients, the number of input declared days must be greater than the model size.

Once the model coefficients are available, the ARXLearner.predict() method is called, which in turn uses the ARXModel class to calculate the actual predictions.

The final step is to save the prediction values into the database via PredictionDB.saveARXPredictions() method.

5.6.2 During-day prediction

The during-day learning and prediction is a more generalized version of the day-by-day prediction model and uses linear regression. The idea is that having more kinds of model attributes, e.g., number of items in particular states, should provide more accurate predictions. In addition, the learning and prediction is performed for multiple time-of-day values (instead of once a day), utilizing past knowledge of the very same time-of-day to predict the end-of-day declared item values.

5.6.2.1 Data source

The input to the learning and prediction is mainly derived from the EVENTS (48), CONSIGNMENT_HISTORY (43) and ITEMS_HISTORY (44) tables.

Data from the EVENTS table are read in event_timestamp order by the MLDB.getMLEvents() method. Unfortunately, joining each event with the consignment and item information degrades performance considerably, therefore, consignment and item information is queried separately through MLDB.getMLConsignments() and MLDB.getMLItems() methods, which return in-memory mappings of the information. When the event sequence is traversed, the appropriate information needs to be looked up from these mappings. (Note that depending on the learning horizon length, storing the mappings may consume large amount of memory, in order of several hundred megabytes.)

5.6.2.2 Attributes

Attributes, the numerical input for the learning and prediction is derived from the data source by using a scheme which defines how and what events to aggregate later on into a single value.

These definitions are maintained by the MLAttribute class, which has the following relevant fields:

```
class MLAttribute {
  ItemEventTypes event;
  int waiting;
  boolean current;
  int day;
}
```

The event selects the event type (see 5.3.5). The waiting has a complicated definition: if less than zero, then the attribute represents those of items that have encountered the event. In addition, if the day is greater than zero, the value needs to be read from the previous day(s). The current plays here as well, indicating that the value is the aggregated number from the

start of the day, or the aggregated number remaining till the end of day. If the `waiting` is greater or equal to zero, it represents those items that haven't reached a specific event type yet in their lifecycle (e.g., not yet declared); and reached their current event type `waiting` days or longer ago.

The `MLAttribute` implements a custom `toString()` method thad produces a human-readable encoding of the attribute definition, which is also used as XML node name in the database-stored models (see section 5.6.2.9).

A few examples are listed in table 78.

CW0	event=CREATED, waiting=0
D	event=DECLARED, waiting=-1, current=true, day=0
CR	event=ENTERED, waiting=-1, current=false, day=0
R2	event=DECLARED, waiting=-1, current=false, day=2

Table 78: Example ML attributes

5.6.2.3 Timepoint aggregation

The logic behind the timepoint aggregation is simple: as time progresses, at predetermined time of days, take a snapshot of the item events and count all sorts of properties of the items (e.g., number of created, number of declared, etc.).

The aggregation is performed by the `MLTimePointAggregator` which is an `Observer<ItemEvent>` and expects a stream of `ItemEvent` records in time order.

The `next()` method updates the `MLItemStateTracker` object and checks if a timepoint has been passed. Item state information older than the `maxWorkdays` is removed at each timepoint, keeping the state concise. The `aggregate()` method locates the consigment details and aggregates the item states via `aggregateState()`.

Once the states have been aggregated, (due to performance reasons) records are saved into a binary representation in the system's temporary directory under the names like `tpi_HHmm_1234567890123.dat` for the inbound- and `tpo_HHmm_1234567890123.dat` for the outbound aggregate (where `HHmm` is the time of the day).

The binary representation's row size depends on the number of elements in the `UnitStatus` enumeration (m), but has the following general structure:

```
day: long; // 64-bit
depot: word; // 16-bit
service_level: byte;
event_type: byte;
```

```
days_type: byte;
value_1: double;
value_2: double;
⋮
value_m: double;
```

And has a row size of `13 + m * 8` bytes. The `service_level` is an ordinal into the `ServiceLevel` enum and the `event_type` is an ordinal into the `ItemEventTypes` enum.

Classes are provided to access these binary timepoint aggregate files: `MLArrayTimePoint`, `MLMappedTimePoint` and `MLRandomAccessTimePoint`, depending on how much memory is available and whether the code is running on a 64-bit system.

The aggregator contains an optional set of blacklisted days (`blacklist`), days where the events are registered, but no timepoint-aggregation is performed. When the `aggregate()` method is running, events on these blacklisted days are interpreted as they were actually on the last regular day, which affects the *waiting*-like calculations. The reason is to exclude low-volume days as inputs to the learning process in order to increase prediction accuracy.

To support the remaining-type attributes, an additional timepoint is added for the time `23:59:59.999` as the end-of-day time point. Remaining values are then computed from `value(end-of-day) - value(now)` fashion.

5.6.2.4 Virtual aggregation

The virtual aggregation is a simple method to decrease the effects of territory changes in the model by narrowing down the historical items considered. The logic works by checking the depot's current territory, and considering only those items which came from or go into the given territory before the current territory took effect. The matching is done via the postcode's `grouping` property.

For example, a depot's current territory from `29/7/2013` contains the group `ABC`. Items after the date are all aggregated, but items before this date is checked. Given two items, P1 `28/7/2013, DEF` and P2 `27/7/2013, ABC`, the item P1 is ignored and P2 is accepted.

The virtual aggregation logic is embedded into the `MLTimePointAggregator.aggregate()` method and can be explicitly enabled or disabled via the `useInboundVirtualAggregation` and `useOutboundVirtualAggregation` fields.

5.6.2.5 Learning process

The learning process is managed by the `MLLearnerTask` class. For maximum flexibility, several parameters are available to guide the process:

Name	Type	Description
days	int	The number of calendar days to aggregate in the timepoint aggregation.
use-blacklist	boolean	Ignore weekends and holidays.
holiday-gap	int	Number of calendar days to ignore after each holiday.
hub	long	The hub filter.
attribute-selection	boolean	Use attribute selection.
max-attributes	int	In attribute selection mode, how many attributes to try.
exact-attributes	string	Comma separater attribute names in the same naming scheme as in 5.6.2.2.
max-workdays	int	Number of working days to consider.
min-timepoint-hour	int	The first hour of day when the timepoint sequence should start.
max-timepoint-hour	int	The last hour of day when the timepoint sequence should end.
timepoint-minute-step	int	The minutes between two timepoints.
exact-timepoints	string	Comma separated list of HH:mm values specifying concrete timepoint hours and minutes.
days-to-learn	int	Number of days to learn, e.g., just today = 1, today and tomorrow = 2, etc.
depots	string	Comma separated list of depot identifiers
service_level_mode	int	Bitflags for each ServiceLevel enum value.
unit_mode	int	Bitflags for each UnitStatus enum value.
direction_mode	int	Bitflags for the transfer direction mode to calculate, e.g., bit 0 = inbound, bit 1 = outbound.
training_split	double	The ratio of the training set to the whole input set.

Table 79: Parameters of the MLLearnerTask

The first step determines the active depots which had any item movement in last max-workdays workdays.

The runTimePointAggregation() prepares the data sources and performs the aggregation for each timepoints via the MLTimePointAggregator. The method then returns the references to the temporary files containing the binary aggregated data.

The second step is performed by the learnModels() method. Depending on the

attribute-selection status, the attribute selection process is run in the MLLearner.withSelectionAll() method. If the exact-attributes is given, the MLLearner.learnAll() method is called. The computation is performed for the allowed service levels, unit-of-measures, directions and days. The learning results are stored in an MLLearnedModels instance.

In the final step, the learned models are saved into the database via the saveLearnedModels() and MLDB.saveMLModel() methods in an XML format (see 5.6.2.9).

5.6.2.6 Linear regression

The learning process uses linear regression to calculate the weights of attributes which fit the observations in the best least-squares manner.

$$\text{minimize } \|Ax - b\|_2^2 \tag{7}$$

Where A is the attribute matrix. Each row of the matrix is a day from the binary data file of the time point, and each column represents an attribute value. The x is the vector of weights the process tries to learn and the b is the target vector, e.g., the value to predict at each day (for example, the end-of-day delcared item count). The attribute matrix and target vector is built by the MLLearner.createMatrixVector() method and is stored in an MLMatrixVector record. The rows of the attribute matrix and target vector is calculated via the MLLearner.setRow() method (which is also used during the prediction phase to create a single row attribute matrix to be multiplied by the weights).

The attribute matrix is constructed with the so-called intercept-term as its last column, or the day-of-week indicator columns (i.e., value in column d_1 is 1 iff the given row's date is a monday, etc.). The rows are set up in time-ascending order.

If the training_split parameter is less than 1, the attribute matrix and target vector is split into separate training (A_{train}, b_{train}) and test sections (A_{test}, b_{test}) (otherwise, the while matrix is used for both training and testing).

The MLLearner.computeMSE() method calculates the x weights and the quality of the learning, such as the mean absolute error (MAE), mean squared error (MSE), the mean and standard deviation of the residual ($A_{test}x - b_{test}$).

The method supports several algorithms to solve (7):

QR decomposition is the fastest method of them all. The logic is that the matrix A_{train} can be decomposed into two matrices Q and R that satisfy $A_{train} = QR$ where Q is orthogonal ($Q^TQ = I$) and R is upper triangular. The calculation is performed by the FastQR class, an

extension to *commons-math*'s `QRDecomposition` class. The speed comes from the equation

$$Rx = Q^T b_{train} \tag{8}$$

which is solved via back-substitution.

SVD is the Singular value decomposition-based pseudoinverse solver which calculates the analytical solution via:

$$x = (A_{train}^T A_{train})^{-1} A_{train}^T b_{train} \tag{9}$$

Usually, the $(A^T A)^{-1}$ can't be calculated directly, therefore a so-called pseudo-inverse is computed via *commons-math*'s `SingularValueDecomposition` class.

Ridge regression is an extension to the SVD ((9)) by adding a compensation term:

$$x = (A_{train}^T A_{train} + kI)^{-1} A_{train}^T b_{train} \tag{10}$$

where k is the ridge factor, usually between $10^{-8} \ldots 10^{-5}$ and I is the identity matrix with the same rank as $A_{train}^T A_{train}$. The function of the extra term is to stabilize the inverse and avoid a singular matrix in many cases.

5.6.2.7 Attribute selection

If the `attribute-selection` is enabled, a greedy algorithm is used to determine the most viable `max-attributes` numer of attributes.

In the first step, the attribute matrix and target vectors are generated for all potential attributes (see 5.6.2.6). The greedy algorithm in `MLLearner.withSelectionAll()` method works as follows:

1. Let the current attribute set be empty.
2. Given a list of available attributes, select a next attribute and add it to the current attribute set.
3. Create a projection from the attribute matrix which contains only those columns which are in the current attribute set.
4. Run the linear regression learning as described in section 5.6.2.6.

5. Once all available attributes have been tried, find the learnt model with the best MSE, fix the attribute in the current attribute set and remove it from the list of available attributes

6. Go back to step 2 until the `max-attributes` have been reached.

5.6.2.8 Prediction process

The learning process is managed by the `MLPredictorTask` class. For maximum flexibility, several parameters are available to guide the process:

Name	Type	Description
`days`	int	The number of calendar days to aggregate in the timepoint aggregation.
`use-blacklist`	boolean	Ignore weekends and holidays.
`holiday-gap`	int	Number of calendar days to ignore after each holiday.
`hub`	long	The hub filter.
`max-workdays`	int	Number of working days to consider.
`min-timepoint-hour`	int	The first hour of day when the timepoint sequence should start.
`max-timepoint-hour`	int	The last hour of day when the timepoint sequence should end.
`timepoint-minute-step`	int	The minutes between two timepoints.
`exact-timepoints`	string	Comma separated list of `HH:mm` values specifying concrete timepoint hours and minutes.
`ignore-run-at`	string	Comma separated list of `HH:mm` values specifying when not to run the prediction.
`days-to-predict`	int	Number of days to predict, e.g., just today = 1, today and tomorrow = 2, etc.
`depots`	string	Comma separated list of depot identifiers
`service_level_mode`	int	Bitflags for each `ServiceLevel` enum value.
`unit_mode`	int	Bitflags for each `UnitStatus` enum value.
`direction_mode`	int	Bitflags for the transfer direction mode to calculate, e.g., bit 0 = inbound, bit 1 = outbound.

Table 80: Parameters of the MLPredictorTask

The first step determines the active depots which had any item movement in last `max-workdays`

workdays, then retrieves the latest models from the database via `MLDB.getMLLatestModels()` method.

The `MLPredictorTask` uses the same `MLLearnerTask.runTimePointAggregation()` to prepare the data sources and perform the aggregation for each timepoints via the classMLTimePointAggregator as mentioned in 5.6.2.5, but in this case, only the last `max-workdays` worth of data is aggregated. The method then returns the references to the temporary files containing the binary aggregated data.

The second step is performed by the `MLPredictorTask.runPredictions()` method, which depending on the parameters specified in table 80, calculates the predictions per timepoint, depot, service level, unit-of-measure, direction and target day.

The prediction calculation reuses the `MLLearner` to setup the attribute matrix for the current day and computes the target value (e.g., remaining to be declared) with a simple matrix multiplication:

$$b = A_{today}x \tag{11}$$

which is a single scalar value. The predicted value, with the additional contextual information (e.g., which depot, timepoint, service level, etc.) is saved in an `MLPrediction` record class.

The final step then saves the predictions via the `MLDB.saveMLPredictions()` method.

5.6.2.9 Model XML format

The during-day learning output is stored as CLOB object in the *ML_MODELS* table (see table 58). Each XML contains models for all timepoints, all service levels, all unit of measures, both inbound and outbound directions. (The reason for a single XML per depot is that storing model details on a column-vise level takes up huge number of rows and due to the large number of columns in the primary key, the storage efficiency is about 5%.)

The basic structure is as follows:

```
<models>
  <inbound>
    <timepoint>
      <unit>
        <model>
          <attribute />
          ⋮
          <day />
          ⋮
          <intercept />
```

```
          <target/>
        </model>
        ⋮
      <unit>
      ⋮
    </timepoint>
    ⋮
  </inbound>
  <outbound>
    <!-- same as inbound -->
  </outbound>
</models>
```

The models has a hub and depot attributes that matches the database record's similar fields. Under the models element, the optional inbound and optional outbound elements are located, both with the same inner structure.

Inside the inbound (or outbound) elements, multiple timepoint elements might be listed, each for every timepoint. Its time='hh:mm:ss.SSS' attribute specifies the time of day.

Each timepoint may contain several unit child elements. Each unit element has a type='string' attribute which refers to the UnitStatus enumeration's names. The day='int' attribute specifies for which day the entire list of child model elements are refering to (e.g., predicting for 0=today, 1=tomorrow, etc.).

The model element has an optional service_level='string' attribute, which refers to the ServiceLevel enumeration's names. If this attribute is missing, the model refers to the all-service-level case. The model also records the learning statistics, such as the Mean Average Error (mae='double'), Mean Squared Error (mse='double'), number of input days (n='int'), residual error-mean (error-mean='double') and standard deviation (error-std-dev='double').

Each model may contain one or more attribute child nodes, each of them has several XML attributes. The event='string' is the name of the ItemEventTypes enumeration value, the waiting='int' indicates the wait mode (e.g., -1=current, 0=waiting at least since today, 1=waiting at least since yesterday, etc.). The day='int' select a previous workday or the current day (=0). The current='boolean' indicates if the attribute represents a value since the start of the day (=true) or the remaining value (=false). The weight='double' is the learnt weight of the given attribute. The name='string' is the user-readable name generated by the naming scheme described in section 5.6.2.2.

The day and intercept are exclusive elements. The day elements refer to the day of week attributes. The name='string' contains the English 3-letter day name . The weight='double' is the learnt weight. The textttintercept simply contains the weight='double' value.

The target selects the target attribute to be learned/predicted. Its structure is similar to the

attribute element's structure, but without the weight attribute.

Parsing and generating such XML is maintained by the MLLearnedModels, MLResult and MLAttribute classes' save() and load() methods.

5.6.2.10 See also

- eu.advance.logistics.live.reporter.prediction.ml.MLConfig
- eu.advance.logistics.live.reporter.prediction.ml.MLConsignment
- eu.advance.logistics.live.reporter.prediction.ml.TimePointFiles

5.7 Scheduler

5.7.1 Overview

The scheduler application finds the best place for unloading and loading each truck and the best sequence for processing those vehicles to utilize transportation resources in an efficient way. Lorries can be dispatched to the warehouse in an order and to a *lorry position* so that *storage areas* fill up as planned and the total travel distance of forklifts is reduced, with a concomitant reduction in the number of times the forklifts get in each other's way. This document describes the main algorithms used for the scheduling routines, the necessary input data and its file structure for producing the output.

The scheduler application consists two similar components: one responsible for finding optimal tipping/loading positions inside the warehouses for the lorries in the queues and one determining the warehouses for each lorry at the hub to enter next. Both components use the same input-output data and file structure. Furthermore there are parts in the computation method which are basically the same in both components.

5.7.2 Architecture

IO operations
These operations are responsible for reading and writing the data described by the inner data model. The read and write operations are transferring data between plain text files and the inner data model.
source files: simulator/IOProcess.h simulator/IOProcess.cpp

Table 81: IO operations

Inner data model
It consists the classes that store all the information the scheduler routines work with. Some of the classes are directly written by the io operations while others are constructed during the data preparation.
source files: simulator/Entities.h simulator/Entities.cpp simulator/Layout.h simulator/Layout.cpp simulator/Model.h simulator/Model.cpp

Table 82: Inner data model

Data preparation
The preparation of the data is done by a few methods which mainly construct an array based data model for improving performance and generating the root vertex of the search tree which describes the initial state of the scheduled system.
source files: simulator/PositionEstimation.h simulator/PositionEstimation.cpp simulator/SequenceEstimation.h simulator/SequenceEstimation.cpp main.cpp

Table 83: Data preparation

Scheduler routines
Scheduler routines are responsible for building up the search tree by generating child vertices starting from the root. The children are generated by varying the value of some decision points until the tree reaches a specific depth. Possible scheduling solutions appear on the leaves of the tree.
source files: simulator/PositionEstimation.h simulator/PositionEstimation.cpp simulator/SequenceEstimation.h simulator/SequenceEstimation.cpp

Table 84: Scheduler routines

Data extraction
The data extraction methods extract specific information from the search tree and put them in the inner data model which than can be written in a text file for further processing.
source files: simulator/PositionEstimation.h simulator/PositionEstimation.cpp simulator/SequenceEstimation.h simulator/SequenceEstimation.cpp

Table 85: Scheduler routines

5.7.3 Components: Input structure

5.7.3.1 General

The input data feeds the scheduler and controls it through some numerical parameters. The data is partitioned into five types: dynamic hub layout description, dynamic lorry description, dynamic depot description, constant parameters and configuration parameters.

The hub layout describes the physical attributes of the hub, the placement and dimensions of warehouses, storage areas and lorry positions and also contains information about the items currently being in the warehouses.

The lorry description lists all the lorries currently being at the hub along with their attributes such as their licence plate, home depot, list of warehouses where they have already been and the items which are still on them.

The depot description lists all the items which have not arrived yet from specific depots. Constant parameters describe a priori knowledge about the system which has been modelled for example the speed of forklifts and time needed to handle items.

Finally the configuration parameters define a few interaction points for the scheduler.

5.7.3.2 File structure

The necessary input data is described by a plain text file. It contains five different blocks, each of them starting with an identifier string (`LAYOUT`, `LORRIES`, `DEPOTS`, `PARAMETERS`, `CONFIGURATION`) and ending with a closing string (`END`). This structure makes it easy to relocate any block to a different file. Currently they are located in the same input file.

The first three blocks (layout, lorries, depots) are dynamic e.g. they can vary in size and they use an element based structure similar to the xml format. The second two blocks (parameters, configuration) are static key-value pairs.

1. **layout** describes the location/orientation of the warehouses of a hub and the item- and lorry positions inside them. The layout of a hub is defined by a text block with the following fields:

   ```
   hub x=<float> y=<float> width=<float> height=<float>
   ```

   one hub element can be defined by giving its:

   - `position`: x and y coordinates (positive real numbers with 0)
   - `width`: (positive real number)
   - `height`: (positive real number)

```
warehouse x=<float> y=<float> width=<float> height=<float>
  angle=<float> forklifts=<int> name=<string>
```

inside a hub element, one or more warehouse elements can be defined by giving their:

- position: relative x and y coordinates inside a hub (positive real numbers with 0)
- width: (positive real number)
- height: (positive real number)
- angle: the rotation of the warehouse (real number)
- forklifts: the number of forklifts working in the warehouse (positive integer)
- name: (string without any whitespace character)

```
lorryposition x=<float> y=<float> width=<float> height=<float>
  enter_time=<int> leave_time=<int> index=<int>
```

inside a warehouse element, lorry position elements can be defined by giving their:

- position: relative x and y coordinates inside a hub (positive real numbers with 0)
- width: (positive real number or negative if side is 1)
- height: (positive real number or negative if side is 3)
- enter_time: time taken to occupy the lorry position (positive integer)
- leave_time: time taken to leave the lorry position (positive integer)
- index: for identification purposes (positive integer with 0)

```
storagearea x=<float> y=<float> width=<float> height=<float>
  depot=<int> capacity=<int> priority=<int> side=<int>
  index=<int>
```

inside a warehouse element, storage area elements can be defined by giving their:

- position: relative x and y coordinates inside a hub (positive real numbers with 0)
- width: (positive real number)
- height: (positive real number)
- depot: from which depot does the storage area accept items (positive integer)
- capacity: maximum number of palettes in the storage area (positive integer)
- priority: type of items in the storage area (0:normal, 1:priority)
- side: on which side of the warehouse the storage area is located (0: left side, 1: right side, 2: entrance side, 3: exit side)

- index: for identification purposes (positive integer with 0)

```
item depot=<int> items=<int> priority=<int>
```

inside a lorry element, item elements can be defined by giving their:

- depot: to which depot does the item belong to (positive integer and there has to be a storage area in the warehouse with the same depot value)
- items: how many items are actually on the floor from this depot (positive integer)
- priority: type of the item (0:normal, 1:priority)

2. **lorries** describes the attributes of the lorries. The lorries at the hub are defined by a text block with the following fields:

```
lorry name=<string> depot=<int> freight=<int> leave_time=<int>
  capacity=<int> current_job=<int> position=<int>
  close_to_warehouse=<string>
```

one lorry element can be defined by giving its:

- name: license plate (string without any whitespace character)
- depot: the home depot of the lorry (positive integer)
- freight: indicates whether the items on the lorry are known or not (0: unknown, 1: known)
- leave_time: the latest time when the lorry should leave the hub (positive integer)
- capacity: (positive integer)
- current_job: the index of the currently executed job (positive integer, less than the number of jobs listed for this lorry)
- position: relative position of the lorry if it is in a queue (positive integer)
- close_to_warehouse: indicates whether the lorry is close to a warehouse (string without any whitespace character)

```
job type=<int> start=<int> end=<int> warehouse=<string> lorryposition=<int>
```

inside a lorry element, one or more job elements can be defined by giving their:

- type: (-1:unknown, 0:tip, 1:load, 2:tip and load, 3:none)
- start: when the job started (string in â€˜yyyy-mm-dd hh:mm:ssâ€™ format)
- end: when the job ended (string in â€˜yyyy-mm-dd hh:mm:ssâ€™ format)

- warehouse: in which warehouse is the job executed (string without any whitespace character)
- lorryposition: at which position is the job executed in the warehouse (0-5)

```
item depot=<int> items=<int> priority=<int>
```

inside a lorry element, item elements can be defined by giving their:

- depot: to which depot does the item belong to (positive integer and there has to be a storage area in the warehouse with the same depot value)
- items: how many items are actually on the floor from this depot (positive integer)
- priority: type of the item (0:normal, 1:priority)

3. **depots** at the hub are defined by a text block with the following fields:

```
depot  number=<int>
```

one depot element can be defined by giving its:

- number: the identifier number of the depot (positive integer)

```
item depot=<int> items=<int> priority=<int>
```

inside a depot element, item elements can be defined by giving their:

- depot: to which depot does the item belong to (positive integer and there has to be a storage area in the warehouse with the same depot value)
- items: how many items are actually on the floor from this depot (positive integer)
- priority: type of the item (0:normal, 1:priority)

4. **parameters** describe the constant parameters of the scheduler:

- forklift_movement_speed: average movement speed of forklifts (positive real number)
- forklift_radius: physical radius of the forklifts (positive real number)
- item_loading_time_at_lorry: time taken to load a item to the lorry (positive integer)
- item_tipping_time_at_lorry: time taken to tip a item from the lorry (positive integer)

- item_loading_time_at_storage: time taken to load a item to the storage area (positive integer)
- item_tipping_time_at_storage: time taken to tip a item from the storage area (positive integer)
- min_lorry_reach_time: time taken to reach the closest warehouse for a lorry (positive integer)
- max_lorry_reach_time: time taken to reach the farest warehouse for a lorry (positive integer)

5. **configuration** describes the control parameters of the search:

- current_time: actual time when the scheduling starts (string in yyyy-mm-dd hh:mm:ss format)
- cycle_of_scheduling: time interval covered by scheduling (positive integer)
- max_forklift_assign: maximum number of occasions to assign forklifts to another lorry during a job (positive integer)
- max_forklift_number: maximum number of forklifts that can be working on a lorry at the same time (positive integer)
- beam_size: beam size of the beam search algorithm (positive integer)

5.7.3.3 Source classes

Reading input file: read operations are static methods of the IOProcess class located in the IOProcess.h and IOProcess.cpp files.

Building inner data model: this process is done by the buildModel method located in the main.cpp file and it initializes an instance of the Model class located in Model.h and Model.cpp files.

5.7.4 Scheduler algorithm

Low-level hub scheduling includes two different components to use. The first component is able to estimate the minimum time needed to process a queue in front of a warehouse by determining the best solution based on a few decision points like how many forklifts should work on a lorry or which lorry position should the entering lorry occupy. This component is referred as PositionEstimation in the document. The second component is able to determine an optimal sequence for the lorries to enter the warehouses by minimising floorspace at the hub, tipping/loading time for the lorries and maximising the number of moved items. This component is referred as SequenceEstimation in the document.

5.7.4.1 Graph based scheduling algorithm

Both components use the same graph based algorithm called beam search. The following steps are executed during one run of the algorithm:

- Generate a root vertex which represents the initial state of the modelled system
- Generate child vertices in the tree, level by level, where a parent-child relation represents that a lorry entered one of the warehouses (the depth of the tree equals the number of lorries that have been scheduled)
- Order the vertices of a given level decreasing by a fitness value before generating children and keep the first n vertices only, where n is the size of the beam
- Stop building the tree at a specified depth
- (extract information from the tree)

Generating the child vertices is controlled by some decision points to cover a wide range of solutions and the fitness value and beam size are responsible for selecting the promising branches only.

5.7.4.2 Measuring tipping/loading time

This calculation forms the computational basis of scheduling.

An estimated time (T_p) of handling a item (p) can be calculated by the following equation (handling a item means a forklift goes to a lorry position or storage area, picks up the item, goes to a storage area / lorry position and puts down the item):

$$T_p = 2t_p + 2r\frac{|P - B|}{v} \tag{12}$$

where

- t_p is the time needed to pick up or put down a item,
- P is the lorry position,
- B is the position of the destination storage area considering its fullness, i.e. the position of the storage area's farthest item from the warehouse's wall,
- v is the velocity of forklifts and r is a heuristic multiplier to simulate the fact that forklifts cannot move straight between points or move at maximal velocity continuously.

An estimated time (T_l) of tipping/loading a lorry (l) can be calculated by the following equation:

$$T_l = \sum_{p \in L} \frac{T_p}{f} \tag{13}$$

where

- L is the set of items which are moved during the tipping/loading and
- f is the number of forklifts assigned to the lorry.

5.7.4.3 Calculate fitness value

The *PositionEstimation* component calculates the fitness value of the vertices by solving the following equations with an LP solver:

$$\forall i, 1 \leq i \leq n : S_i - S_{i-1} \geq \min_{0 \leq j < m} (t(j)) \tag{14}$$

$$\forall i, m \leq i \leq n : S_{i-m} + \sum_{j=0}^{l-1} P_{i-m,j} x_{i-m,j} \leq S_i \tag{15}$$

$$\forall i, m-1 \leq i < n : \sum_{j=0}^{l-1} \sum_{k=0}^{m-1} (j+1) x_{i-k,j} \leq r \tag{16}$$

$$\forall i, 0 \leq i < n : \sum_{j=0}^{l-1} x_{i,j} = 1 \tag{17}$$

where

- $S_i \in \mathbb{N}$ is the time when the i^{th} lorry enters the warehouse,
- $n \in \mathbb{N}$ is the length of the queue,
- $m \in \mathbb{N}$ is the number of lorry positions in the warehouse,
- $t : \mathbb{N} \to \mathbb{N}$ is a function which maps the indexes of lorry positions in the warehouse to the amount of time (in seconds) needed to occupy them,
- $P_{i,j} \in \mathbb{N}$ is the time needed to tip and/or load the i^{th} lorry with j forklifts working on it.
- l is the maximum number of forklifts working on a lorry simultaneously

- r is the number of forklifts in the warehouse and
- $x_{i,j}$ is an indicator variable to tell whether j forklifts are working on the i^{th} lorry or not. For the sake of simplicity, the $x_{i,j}$ variable is not discreet so actually it describes the proportion of j forklifts working on the i^{th} lorry.

The *SequenceEstimation* component uses a simple heuristic equation for determining fitness values for vertices:

$$f = \frac{\Delta t}{T} \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m_i-1} \frac{s_{i,j}}{S_{i,j}}}{\frac{\sum_{k=0}^{l-1} p_k}{l}} \tag{18}$$

where

- Δt is the elapsed time from the start of scheduling represented by the given vertex,
- $n \in \mathbb{N}$ is the number of warehouses at the hub,
- $m_i \in \mathbb{N}$ is the number of storage areas in the i^{th} warehouse,
- $s_{i,j}$ and $S_{i,j}$ are the fullness and the capacity of the j^{th} storage area in the i^{th} warehouse represented by the given vertex,
- $l \in \mathbb{N}$ is the number of lorries scheduled so far and
- p_k is the number of items that were moved during the execution of the k^{th} lorry's job.

5.7.4.4 Data structures

The `PositionEstimation` component uses the following vertex structure in its search tree:

PositionEstimation::TreeNode
cl : int*
af : unsigned char**
ra : unsigned char*
st : long*
je : long*
n : unsigned
r : unsigned
globalTime : long
fitness : long
parent : TreeNode*

Table 86: PositionEstimation::TreeNode structure

where

- `cl`: the indexes of the lorries occupying the lorry positions in the warehouse or -1 if the lorry position is empty
- `af`: the number of forklifts working at a lorry position after each reassignment
- `ra`: the number of reassignments at a lorry position
- `st`: the time when the jobs start at the lorry positions in the warehouse or -1 if the lorry position is empty
- `je`: the time when the jobs end at the lorry positions in the warehouse or -1 if the lorry position is empty
- `n`: number of lorry positions in the warehouse
- `r`: maximum number of forklift reassignments per lorry
- `globalTime`: global time to indicate when the next lorry can enter the warehouse
- `fitness`: the fitness value of this vertex
- `parent`: the parent vertex of this vertex

The `SequenceEstimation` component uses the following vertex structure in its search tree:

SequenceEstimation::TreeNode
cl : int**
st : long**
je : long**
pp : unsigned short**
ep : unsigned short**
pt : unsigned short*
et : unsigned short*
ps : char*
vs : char*
gt : long*
ns : unsigned*
m : unsigned
s : unsigned
l : unsigned
lorry : int
fitness : float
movedItems : unsigned
currentJob : char
nextJob : char
parent : TreeNode*

Table 87: SequenceEstimation::TreeNode structure

where,

- `cl`: the indexes of the lorries occupying the lorry positions in the warehouses or -1 if the lorry position is empty
- `st`: the time when the jobs start at the lorry positions in the warehouses or -1 if the lorry position is empty
- `je`: the time when the jobs end at the lorry positions in the warehouses or -1 if the lorry position is empty
- `pp`: number of priority items at the hub per depot per warehouse
- `ep`: number of normal items at the hub per depot per warehouse

- pt: number of priority items on the lorry that is represented by this vertex per depot
- et: number of normal items on the lorry that is represented by this vertex per depot
- ps: indicates which are the possible warehouses for the lorries to visit
- vs: indicates which are the warehouses that the lorries have already visited
- gt: global time for each warehouse to indicate when the next lorry can enter the warehouse
- ns: number of lorry positions in the warehouses
- m: number of depots
- s: number of warehouses
- l: number of lorries
- lorry: the index of the lorry represented by this vertex
- fitness: the fitness value of the vertex
- movedItems: total number of moved items on the route from the root vertex to this vertex
- currentJob: indicates the type of the job that was executed on the lorry represented by this vertex
- nextJob: indicates the type of the job to be executed next on the lorry represented by this vertex
- parent: the parent vertex of this vertex

5.7.4.5 Source classes

Graph generating: building the search tree is done by the computeChoices, generateRoot, generateChildren, processLevel and addNodeToLevel methods of the PositionEstimation/SequenceEstimation class located in PositionEstimation.h and PositionEstimation.cpp/SequenceEstimation.h and SequenceEstimation.cpp files.

Calculating tipping/loading time: this calculation is done by the executeJob method of the PositionEstimation/SequenceEstimation class located in PositionEstimation.h and PositionEstimation.cpp/SequenceEstimation.h and SequenceEstimation.cpp files.

Calculate fitness value: this value is calculated by the calculateFitness method of the PositionEstimation/SequenceEstimation class located in PositionEstimation.h and PositionEstimation.cpp/SequenceEstimation.h and SequenceEstimation.cpp files.

Vertex structures: vertex structure is described by the TreeNode class inside PositionEstimation/SequenceEstimation class located in PositionEstimation.h and PositionEstimation.cpp/SequenceEstimation.h and SequenceEstimation.cpp files.

5.7.5 Output structure

5.7.5.1 General

The output data is produced by the scheduler and contains a dynamic lorry description only.

The lorry description lists all the lorries currently being at the hub along with their attributes such as their licence plate, home depot, list of warehouses they have to visit and the items which will be on them after finishing the scheduled jobs.

5.7.5.2 File structure

The output data is described by a plain text file. It contains one block only starting with an identifier string (LORRIES) and ending with a closing string (END). It is completely identical to the corresponding block of the input.

1. **lorries** describe the attributes of the lorries. The lorries at the hub are defined by a text block with the following fields:

```
lorry name=<string> depot=<int> freight=<int> leave_time=<int>
  capacity=<int> current_job=<int> position=<int>
  close_to_warehouse=<string>
```

one lorry element can be defined by giving its:

- `name`: license plate (string without any whitespace character)
- `depot`: the home depot of the lorry (positive integer)
- `freight`: indicates whether the items on the lorry are known or not (0: unknown, 1: known)
- `leave_time`: the latest time when the lorry should leave the hub (positive integer)
- `capacity`: (positive integer)
- `current_job`: the index of the currently executed job (positive integer, less than the number of jobs listed for this lorry)
- `position`: relative position of the lorry if it is in a queue (positive integer)
- `close_to_warehouse`: indicates whether the lorry is close to a warehouse (string without any whitespace character)

```
job type=<int> start=<int> end=<int> warehouse=<string> lorryposition=<int>
```

inside a lorry element, one or more job elements can be defined by giving their:

- type: (-1:unknown, 0:tip, 1:load, 2:tip and load, 3:none)
- start: when the job started (string in yyyy-mm-dd hh:mm:ss format)
- end: when the job ended (string in yyyy-mm-dd hh:mm:ss format)
- warehouse: in which warehouse is the job executed (string without any whitespace character)
- lorryposition: at which position is the job executed in the warehouse (0-5)

```
item depot=<int> lifts=<int> priority=<int>
```

inside a lorry element, item elements can be defined by giving their:

- depot: to which depot does the item belong to (positive integer and there has to be a storage area in the warehouse with the same depot value)
- lifts: how many lifts are actually on the floor from this depot (positive integer)
- priority: type of the item (0:normal, 1:priority)

5.7.5.3 Source classes

Extracting data from graph: data extraction is done by the extractSolution methods of the PositionEstimation/SequenceEstimation class located in PositionEstimation.h and PositionEstimation.cpp/SequenceEstimation.h and SequenceEstimation.cpp files.

Writing output file: write operations are static methods of the IOProcess class located in the IOProcess.h and IOProcess.cpp files.

5.7.6 Running the scheduler

The scheduler is a 32-bit native console application, therefore, it can be started from command line. The program has three types of switches which are:

- -c<component name>: tells the scheduler which component to use, possible values are: position (for PositionEstimation component) and sequence (for SequenceEstimation component),
- -i<input file>: tells the scheduler the location of the input file and
- -o<output file>: tells the scheduler the location of the output file.

The -c and the -i switches have to be specified for a successful run of the scheduler. The use of -o switch is optional and if an output file is not specified, the scheduler overwrites the input

file with the output information. Multiple use of any of the switches results in an error message and the scheduler stops.

Examples for running the scheduler:

```
AdvanceDispatcher -cposition "-i./scheduler_input.dat"
AdvanceDispatcher -csequence "-i./scheduler_input.dat"
  "-o./scheduler_output.dat"
```

In the first example the *PositionEstimation* component starts and overwrites the `scheduler_input.dat` file. In the second example the *SequenceEstimation* component starts and the output is written in the `scheduler_output.dat` file.

Minimal system requirements are:

- 1GB RAM
- 2 GHz processor
- 50 Mb free space

List of Figures

List of Tables

Index