Sensor Web Enablement

How-To for developing plugins for a 52°North SPS version 1-00-00

Document Change Control

Revision Number	*Date Of Issue*	*Author(s)*	*Brief Description Of Changes*
0.0	2008-02-13	Henning Bredel	draft version
0.1	2008-03-20	Henning Bredel	section 5 revised

Editors

Henning Bredel

Institute for Geoinformatics
Weseler Straße. 253
D – 48151 Muenster

Phone: +49 (0)251 - 83 300 98
Email: henning.bredel@uni-muenster.de

Licence

Contact: Andreas Wytzisk,

52North Initiative for Geospatial Open Source Software GmbH,
Martin-Luther-King-Weg 24,
48155 Muenster, Germany,
info@52north.org

Table of Contents

1 Introduction

1.1 Scope

This document describes in short how to develop a plugin for a sensor which shall be tasked by a 52°North Sensor Planning Service (SPS).

An SPS is not very functional without any plugin registered to it. So it is a good idea to read this manual first to know which role a plugin plays within an SPS.

1.2 What this document describes

This HowTo describes what you have to consider if you plan to develop a plugin for an SPS. It will give you an idea of what makes up an SPS plugin and which steps are necessary to develop a plugin for an SPS. Furthermore it lists some steps to follow concerning the development.

1.3 What this document does not describe

This document neither gives a complete description nor guidance on how to install or set up an SPS. This file is contained in the SPS service distribution.

This document also does not give a complete manual for implementing the tasking logic of a specific sensor. The fact that there are so many different kinds of sensors available makes it impossible to offer *the one* manual for developing a plugin for a sensor (*Note that the SPS was designed to act as a framework where you can add your plugin to and task it using the SPS interface specified in its corresponding OGC specification you can review here*:

http://portal.opengeospatial.org/files/?artifact_id=23180&version=1

1.4 Some additional words

The SPS framework consists of three components: the AssetManager (AM), the ProfileManager (PM) and the Controller. Together they form the SPS. Several service operations have been defined for the AM and PM to support managing the 52°North SPS.

If you are the administrator of the SPS , you can use some operations (currently only) defined by the AM and PM to administrate your plugin. We get back to this point later, but if you are familiar with the SPS specification do not be confused when you find some more operations which are not defined in the specification.

The SPS in the current version offers you the option also to register your plugin at a Web Notification Service (WNS) when you register it at the SPS. This provides the functionality to receive asynchronous messages via WNS. Since the WNS is currently not yet an official standard of the OGC the SPS uses the current best practice version 0.0.9.

This HowTo currently does not provide more information on how to use the WNS inside a plugin.

You can find more information by looking at the code of the AXIS(tm) camera plugin. You can download it from

http://52north.org/download/sensorweb/sps/52N-swe-axisplugin-1.0.0.zip

1.5 Naming conventions

A plugin defines an installed class of a concrete sensortype. Each instance of this class has

status and behaviour defined by this class. The installation process of a plugin is very easy, since it is done through a simple mapping (from sensor type to implementing class). Refer to the examples in the *HowTo-SPS_install.pdf*.

When we speak of a sensor instance (or shortly: sensor), we mean one of possibly many registered sensors registered to the SPS.

Data types and source code will be written in another font than normal text. The source code written in a code block will be written as its own block in a code style like the following:

Listing 1: Example

```
public class Foo {

      private Foo foo;

      public Foo() {
          super();
          System.out.println("Foobar");
      }

      ... methods etc ...
}
```

In addition to that you can differentiate `data types` from normal font; it comes without serifs and with expanded spacing.

When referring to `xsd`-schema types there should always be a self-explaining namespace abbreviation like `sps:InputDescriptor` or `n52sps:AMConfiguration` before the concrete type (Note here that there are schemata defined by the OGC-SPS specification and 52n service specific schemata). Java types/classes in text appear the first time with their full package path. After that only the classname will be shown (first time: `org.n52.sps.common.SPSSensor`, after that only: `SPSSensor`).

2 Requirements

To develop a plugin for the SPS framework we recommend to use eclipse. The reason is not because we think it is the best IDE but there is an existing target ant task you can use to deploy the downloaded SPS distribution as eclipse WebToolsProject (WTP). For more information consult the *HowTo—Install_SPS*.

The complete list of requirements are the following:

- An installed Java JDK 1.5 or higher [tested with JDK1.6]
- The ant build tool (e.g. as standalone tool or as an integrated one – maybe in eclipse) [tested with ant 1.7.0]
- A runnable SPS framework deployed as WTP project (developer version)

A “Runnable SPS framework” means that you have done the installation/deploy process and made the appropriate settings for service and database. Make sure you have done the initial setting steps described in *HowTo—Install_SPS*.

3 The framework

To develop a plugin for the SPS requires to know how the framework works in principle. This section gives a short overview of the SPS service components, their responsibilities in the framework and how a plugin is involved in the framework.

The SPS is structured in three modules each having its own field of activity. Functions for which a module is not responsible for, will be delegated to the right module or plugin instance via request forwarding. A short description about the fields of activity for each module:

AssetManager (AM): Administers the registered plugins. Is responsible for registration/unregistration (also handles registration of plugins at and communication via WNS). It delegates tasking requests to a plugin.

ProfileManager (PM): Administers the framework specific and sensor specific information. Furthermore the PM is responsible for the sensorML decription of a plugin.

Controller (Control): Acts as a frontend for a user and for the AM/PM and validates (if set in the properties while deploying) all incoming XML-requests, so these must not be validated again.

4 Sensor interface and its operations

This section gives an overview of the sensor interface the SPS offers. Currently two examples of plugins exist that show how plugins can be developed.

4.1 Overview

The plugin pattern makes it possible to register various types of taskable sensors and make them accessible through the SPS framework. To ensure this, you need to keep the following in mind when developing your own plugin:

First, define the *parameters* your sensor needs to be tasked. This is very individually to each plugin but should be the same for each sensor instance registered as such a plugin. These parameters can be defined as a list of `sps:InputDescriptor` but can also be hard coded.

The second thing is that your plugin must inherit from the abstract superclass `Sensor` which acts as a uniform interface for each sensor. The task execution for a sensor is done by the logic contained in the implementation of your plugin. From the parameters given in every task request you can construct your commands and send them to the real sensor.

4.2 The `sensor` interface

To ensure that the SPS can deal with the arbitrary complexity of sensors the developer has to implement a well defined interface. With this you can rely upon the framework and the handlers which will automatically be provided to each plugin instance. Such an instance will be created when a new collection asset is registered at the service.

Note that without a registered plugin the framework is not functional. It will become fully functional—i.e. OGC SPS enabled clients can make use of it—if at least one plugin instance is registered at the AM.

Two example plugins have been implemented to exemplarily show how a plugin implementation looks like:

CITE test plugin: The CITE test plugin was developed during the OGC-OWS5 *compliance test & interoperability evaluation* for the 52North SPS reference implementation. It is comparatively simple and in fact only simulates to task a real sensor. The sensor was defined and the plugin implemented to get the SPS functional for testing. It should be contained by the distribution. If not, please send an email to info@52north.org.

AXIS(tm) plugin: The AXIS(tm) plugin comes with more complexity and tasks an AXIS(tm) PTZ camera as real sensor. It was developed by Johannes Echterhoff and should be available at http://www.52north.org.

4.3 SensorConfiguration (#any)

To make the plugin configurable you can use the `SensorConfiguration` element when you register a sensor at the SPS. Its type is set to `#any`, so you can add any information you think you need to configure the sensor. During the registration process (initiated by sending a Register request to the AM) the AM will forward the document contained in the `n52sps:SensorConfiguration` to the `configure()` method of the instance of the sensor. Parse the configuration document, configure the sensor and set its state as required (see the source code of the example plugins).

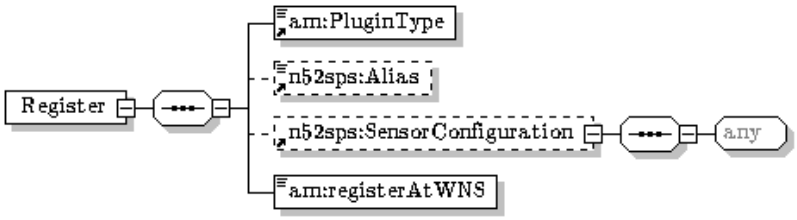

Abbildung 4.1.: The schema of an amRegister request.

In addition to this, a sensor instance has to provide information about the location or area that it is able to gather information from, data services from where the retrieved data could be retrieved and which properties are observed by the sensor. This information will be stored in the SPS in the sensors profile. They could be contained in the sensors configuration file as well.

We recommend to create a well defined datamodel for the sensors configuration file as an XML-schema file. Have a look at the CITE example plugin where the schema `TestPluginConfiguration.xsd` was defined. It describes the datamodel of the sensor which can be used to define an individual configuration for a registration of a new sensor instance.

This is *not* the mandatory way to configure the sensor (since you could also hardcode the information in your plugin code). But we recommend to use this, since you are able to change a sensors configuration through an update. In addition you can use existing elements such an `sps:InputDescriptor` to define the parameters and their constraints for each sensor instance individually.

4.4 Operations

This section gives you more detailed information about the mandatory and optional operations of the sensor interface. In addition to the operations defined in the SPS specification an SPS

can offer more operations, e.g. for administrating the service (registering a plugin or removing it).

Section 4.4.1 lists all operations (mandatory and optional) described in the SPS specification. The following section 4.4.2 describes all additional operations for administrating the plugin via the SPS framework components. Section 4.4.3 descibes methods an asynchronous message handling.

4.4.1 SPS Operations

Each sensor plugin must inherit the supertype `org.n52.sps.common.SPSSensor`. An `SPSSensor` may be any component that is taskable, i.e. that can perform the sps-operations:

- `sps:GetFeasibilityRequest` (optional)
- `sps:SubmitRequest` (mandatory)
- `sps:UpdateRequest` (optional)
- `sps:GetStatusRequest` (optional)
- `sps:CancelRequest` (optional)

(The `sps:DescribeResultAccessRequest` will be handled through the SPS itself.)

All these operations are described in the SPS specification in detail. Here only the default behaviour is shown. If needed, a short additional note is given.

The SPS delegates the requests listed above to the sensor instance. Therefore it is neccessary that your plugin must extend the abstract superclass `SPSSensor`. You can override (or not, depending on whether the operation is a mandatory SPS operation or not) these methods to implement the concrete behaviour of the plugin.

To avoid confusion, note the difference between service-operations (like `n52sps:amRegister`, `n52sps:amGetSensorStatus`, `n52sps:amUnregister` or `n52sps:amUpdateSensorProfile`, etc.) and sps-operations:

The service-operations are defined to manage the SPS framework. They give the SPS-operator an interface to administer the service, its plugins and sensor instances. These operations are *not* part of the SPS specification of the OGC.

Each sensor instance of a plugin will be instantiated through one of the central elements of the SPS, the `org.n52.sps.am.AMResourceContainer` via reflection. So it is neccessary to implement the super constructor in the form presented in listing 2.

Listing 2: Constructor of SPSSensor

```
public class MySensor extends SPSSensor {

      // ... field declarations ...

      public MySensor(String externalSensorID, TargetIDGenerator
                        idGenerator, PMAgent agent,
                        WNSHandler wnsHandler) {

            super(externalSensorID, idGenerator, agent, wnsHandler);

            // ... for configuring use the configure()
      }

      // ... methods etc ...
}
```

Do not mind the `externalSensorID`, `idGenerator`, `agent` or the `wnsHandler` parameters. They will be provided automatically and will not affect the implementation of the plugin itself. You use them whenever you want to create a taskID or if you want to delegate some work to the PM, e.g. you want to update the sensors profile or make a new `n52sps:TaskDataAccess`. The `SPSSensor` class provides a `getExternalSensorID()` method to retrieve the external sensorID (the alias of the sensor instance which is concatenated with the sensorID provider part parameter (set by the SPS provider) as prefix. If set correctly through the SPS provider the `externalSensorID` should be a qualified URN such as `urn:ogc:object:sensor:IFGI:axisCam1`).

The following subsections show you the methods more detail, separated into the mandatory and optional ones.

4.4.1.1 Mandatory operations

Actually there is only the `submit` operation which comes from the SPS specification and has to be implemented by the sensor.

Submit

The parameter contains the `sps:SubmitRequest` sent by the user. Here you can extract the `sps:sensorParams` and also the `sps:notificationTarget` (the registered sensor instance at a WNS—maybe this feature is optional in the next version of the SPS specification).

Listing 3: `submit`

```
public abstract SubmitRequestResponseDocument submit(SubmitDocument sd)
throws ServiceException;
```

Within the `sps:Submit` you can extract the mappings from parameterID to the value array from which you can construct the task command for the sensor.
Note that if these parameter values are of type `#any` you have the highest flexibility. If the parameters are of type `swe:simpleType` (except the ranges and `swe:text`) or `swe:positionType` you can use the `org.n52.sps.common.InputResolver` to resolve the parameters from the given sps:`InputParameter` array.

4.4.1.2 Optional operations

The optional operations have been implemented with standard return values, so if you do not intend to use them you do not have to override them. The default behaviour of the operations can be catched in the listings below.

The listings are a one-to-one copy of the code of `SPSSensor`. It can be reviewed as `javadoc` as well. A short description is given to each. Mostly similar to the `javadoc` comments, too.

getStatus

Like `update` the default behaviour is to return an `ogc:ServiceException` that contains an OGC Exception with code `NoApplicableCode` as well as a textual description which indicates that the operation is not supported by the sensor.

Listing 4: getStatus

```
public GetStatusRequestResponseDocument getStatus(GetStatusDocument
gsd) throws ServiceException {

      // default behaviour:
      ServiceException se = new ServiceException();

      se.addCodedException(
                  ServiceException.ExceptionCode.NoApplicableCode,
                  null,
                  "The GetStatus operation is not supported by this" +
                  " sensor.");
      throw se;
}
```

Update

The default behaviour is to return an `ogc:ServiceException` that contains an OGC Exception with code `NoApplicableCode` as well as a textual description which indicates that the operation is not supported by the sensor.

Listing 5: update

```
public UpdateRequestResponseDocument update(UpdateDocument ud)
                  throws ServiceException {

      // default behaviour:
      ServiceException se = new ServiceException();
      se.addCodedException(
                  ServiceException.ExceptionCode.NoApplicableCode,
                  null, "The Update operation is not supported by"
                   + " this sensor.");
      throw se;
}
```

cancel

Default behaviour is to return an `ogc:ServiceException` that contains an OGC Exception with code `NoApplicableCode` as well as a textual description which indicates that the operation is not supported by the sensor.

Listing 6: CancelRequest

```
public CancelRequestResponseDocument cancel(CancelDocument cd)
                  throws ServiceException {

      ServiceException se = new ServiceException();

      se.addCodedException(
                  ServiceException.ExceptionCode.NoApplicableCode,
                  null, "The Cancel operation is not supported by"
                              + " this sensor.");
      throw se;
}
```

getFeasiblitiy

Default behaviour is to return an `ogc:ServiceException` that contains an OGC Exception with code NoApplicableCode as well as a textual description which indicates that the operation is not supported by the sensor.

Listing 7: `getFeasibility`

```
public GetFeasibilityRequestResponseDocument getFeasibility(
                  GetFeasibilityDocument gfd) throws ServiceException {

      // default behaviour:
      ServiceException se = new ServiceException();
      se.addCodedException(
            ServiceException.ExceptionCode.NoApplicableCode, null,
            "The GetFeasibility operation is not supported by this
            + " sensor.");
      throw se;
}
```

getSensorConfiguration

The SPS where the sensor instance will be registered in will use this method to retrieve an xml encoded configuration file. This file contains all information necessary to restore the current state of this sensor.

As mentioned in section 4.3 (SensorConfiguration) the format of the configuration file is up to the implementer of the plugin. The file could contain full configuration information or just point to where to get more information (e.g. the location of the classes properties file or a database).

This method will most probably be called before the SPS is shut down so that the services state can be persisted and be restored at next startup.

Listing 8: `getSensorConfiguration`

```
public SensorConfigurationDocument getSensorConfiguration() {

      // default behaviour is to:
      return null;
}
```

4.4.2 Service Operations

The following operations are used for managing a registered sensor instance at the AM component.

4.4.2.1 Mandatory operations

Mandatory operations ensure a clean run of the service They define methods which gives access to the sensor instance from outside. While `configure` is called automatically when instantiating the sensor, `getSensorInformation` is requested when the sensor must be registered to the PM.

configure

Although the plugin implementor can choose to hard code the status of a plugin when it is initialized through the `AMResourceContainer` during registration or service restart.

We favor to define a well formed datamodel for the plugin given as an `xsd`-schema file. Such a datamodel as base structure could contain anything you want (The last status of a sensor, the defined input parameters needed to task the sensor, defined constraints such as camera angles which are forbidden, etc.).

Each implementing class must overwrite this method and guarantee that the configuration was successful if no exception was thrown.

Listing 9: configure

```
public void abstract void configure(SensorConfigurationDocument
                                        configXML)
                  throws SensorConfigurationException;
```

getProfileInformation

Use this method to get the ProfileInformation for this sensor. The AM will use this method when a new sensor has been added to the service in order to register its SensorProfile at the PM if this has not been done before (maybe the sensor is just reinitiated after a service shutdown).

Listing 10: configure

```
public abstract ProfileInformationDocument getProfileInformation();
```

4.4.2.2 Optional operations

The optional operations can be implemented to expand the sensors functionality and flexibility. There are several methods to influence tasks and behaviour for each sensor instance.

prepareShutdown

This operation is called from the service when it is about to be stopped. Each sensor which is implementing this method will get the chance to store status, current tasks, etc. before the service stops.

The sensor is responsible to block the creation of new tasks (i.e. to deem all incoming feasibility requests as not feasible, as well as all incoming submit requests) and delay all ongoing tasks when this method has been called. The sensor shall state "SPS service was shut

down" when notifying the user of a task that is about to be delayed now.

The sensor should release all resources that might remain in memory if not being explicitly released (e.g. threads that have been started and are not daemons).

Listing 11: `prepareShutdown`

```
public void prepareShutdown() {

      // default behaviour is to do nothing
}
```

updateSensor

Besides the sps `update` operation (see page 11) described above, the framework offers you another method to update the sensor status at run-time (so you do not have to stop the whole service and update the configuration of the sensor manually). Again, how the information contained in the `SensorConfigurationUpdate` has to look like is up to you, providing you every means to completely update the whole sensor instance or only in parts.

Listing 12: `updateSensor`

```
public SensorConfigurationDocument
updateSensor(SensorConfigurationUpdate
                  scu) throws SensorConfigurationException,
                              PMOperationException {
      return null;
}
```

getSensorStatus

Used to receive status information about this sensor. What you get is the status information which contains at least a list of all tasks known to the sensor and their status.

Additional information can be added which is specific for the current type of sensor. If the sensor does not save any information about incoming tasks, then the returned value may also be `null`, indicating that no status information is available.

The default behaviouris shown in listing 13.

Listing 13: `getSensorStatus`

```
public SensorStatusDocument getSensorStatus() {

      // default behaviour is to:
      return null;
}
```

cancelAllTasks

In contrast to the sps `cancel` operation (see page 12) this method and the following `cancelTasks` method allow the administrator to cancel some or all tasks of a given sensor instance. You can also provide a description about the reason the task was cancelled which should be included in the cancellation message sent to the task owner.

The default behaviour can be overseen in the listing 14.

Listing 14: cancelAllTasks

```
public void cancelAllTasks(StringOrRefType desc) throws
                  CancelTaskException {
      // default is do nothing
}
```

cancelTasks

This method allows administrators to cancel only the tasks referenced by the given set of task IDs.

The default behaviour is shown in listing 15.

Listing 15: cancelTasks

```
public void cancelTasks(Set<String> tasksToDelete, StringOrRefType
desc)
                  throws CancelTaskException {
      // default is do nothing
}
```

getAvailableFeasibilityIDs

Use this method to determine which cached feasibility request data is currently available. The service will use this method at startup to initialize the RequestMapper.

Note: again, this method is optional, so you do not need to override it. The returned `null` value will be interpreted as that the sensor does not support the caching of feasiblitiy ids.

Listing 16: `getAvailableFeasibilityIDs`

```
public Set<String> getAvailableFeasibilityIDs() {
      return null;
}
```

getAvailableTaskIDs

Use this method to determine which tasks are currently available. The service will use this method at startup to initialize the RequestMapper.

Note: again, this method is optional, so you do not need to override it. The returned `null` value will be interpreted as that the sensor does not support the caching of feasiblitiy ids.

The default behaviour can be overseen in the listing 17.

Listing 17: `getAvailableTaskIDs`

```
public Set<String> getAvailableTaskIDs() {
      return null;
}
```

4.4.3 WNS operations

################## DRAFT TO BE DONE ####################

WNS OPERATIONS

Listing 18: handleCommunicationMessage

```
public void handleCommunicationMessage(String serviceType, String
                  serviceTypeVersion, URL serviceURL, String corrID,
                  URL callbackURL, Node payloadNode) {
      // default behaviour is to do nothing
}
```

Listing 19: handleCommunicationMessage

```
public void handleCommunicationMessage(String serviceType, String
                  serviceTypeVersion, URL serviceURL, String corrID,
                  URL callbackURL, SPSMessage message) {
      // default behaviour is to do nothing
}
```

Listing 20: handleCommunicationMessage

```
public void handleMessage(XmlObject message) {
      // default behaviour is to do nothing
}
```

Listing 21: handleNotificationMessage

```
public void handleNotificationMessage(String serviceType, String
                  serviceTypeVersion, URL serviceURL, Node payloadNode)
{
      // default behaviour is to do nothing
}
```

Listing 22: handleNotificationMessage

```
public void handleNotificationMessage(String serviceType, String
                  serviceTypeVersion,URL serviceURL,SPSMessage message)
{
      // default behaviour is to do nothing
}
```

Listing 23: `handleReplyMessage`

```
public void handleReplyMessage(String corrID, Node payloadNode) {
      // default behaviour is to do nothing
}
```

Listing 24: `handleReplyMessage`

```
public void handleReplyMessage(String corrID, SPSMessage message) {

            // default behaviour is to do nothing
      }
```

5 Plugin Development

5.1 Getting started

This section intends to describe the steps which you have to follow when preparing your environment for plugin development.

Alternatively you may find some more information at the Twiki page for developing plugins for the SPS:

https://52north.org/twiki/bin/view/Sensornet/SpsPlugins

5.1.1 Deploy the SPS as developer version in your eclipse WTP workspace

This is needed to let the SPS run within eclipse. You can leave the SPS code as is.

Refer to the *HowTo-Install_SPS.pdf* if there isn't an SPS instance in your workspace. The document comes with the SPS distribution.

5.1.2 Create a plugin project

Keep the SPS and the plugin projects seperated from each other. When doing so, you can distribute the plugin without SPS easily, afterwards.

To run the plugin in combination with the SPS, the plugin must offer itself and its runtime libraries to the SPS. On the other hand, the SPS should offer its runtime libraries to the plugin, so the plugin can use them in the same way as it lays within the framework. This way simulates the later runtime environment in a tomcat container.

To avoid a circular build path you have to create a user library containing the SPS libs, instead of declaring the SPS as required project on the build path. In the Java Build Path properties of the plugin create a new user library and add the libraries of the SPS needed to build and run the plugin. On the Libraries tab click

```
Add Library => User Library => next => User Libraries...=> New... (type
               name, e.g. sps-libs) => ok => Add JARs...
```

Browse to the WebContent/WEB-INF/lib folder of the SPS framework in your workspace. Add all the libraries within the folder and click ok. Finish the wizard.

Now let the SPS know there is a project which can be used at runtime. This is important, because the SPS only knows things on buildpath which lay within the WEB-INF folder while

running as webapplication. Set the dependency within the SPS properties: Use the `JEE Module Dependencies` tab and select the plugin project there. Now the SPS can resolve the needs of the plugin during runtime.

Note: If the plugin uses its own libraries at runtime, these must be declared as `JEE Module Dependency`, too. Add and select them as you did with the plugin project in the SPS properties.

The plugin must extend the `SPSSensor` interface so the SPS can use it. Copy the `52nSpsCommon-v100.jar` to the lib folder of your plugin and add it to the build path. This library must not be imported into the SPS module dependencies.

From now on you can start the development.

5.2 Define a configuration schema for the plugin

You are free to define any kind of configuration your plugin may need. When you register the plugin at the SPS you send a concrete configuration for a sensor instance within the register document and it will be delegated to the plugin when instantiating it (see the configure method).

We recommend to define parameters the plugin needs to be tasked beforhand. Use the SweCommon data types for that. These can be put into an `sps:InputDescriptor` with a unique `parameterID`. This `parameterID` will be used to refer to that `sps:InputDescriptor` later when submitting parameter values via a `Submit` request to task the sensor instance.

5.3 Developing

Create a class which extends the `SPSSensor` interface from the SPS framework. Click

```
Source => Override/Implement Methods...
```

and choose the methods, your plugin wants to support. Refer to the JavaDoc or read section 4.4 which methods have to be implemented.

Keep in mind that the sensor must hold a valid status during runtime, so that it can respond a correct report for a `sps:GetStatusRequest`. It might be helpful to create a state chart of the plugins behaviour before starting to implement the plugin. However, plugins may also choose to 'forget' the status of finished tasks, responding with an exception with code `InvalidTaskID`.

For checking and resolving the input parameters you can use the `InputResolver` class in the `common` package. The operations are based on SweCommon types. The parameters will be checked against their type and constraints (if there are some) defined in the appropriate `sps:InputDescriptor`. For future releases the SPS will be able to perform syntax checking for you.

5.4 Deploying

When you have finished developing the plugin, you can archive it as a `jar` and copy/paste it into the SPS lib folder. Restart the service and begin the install/registration via the administration interface of the SPS. Refer to the examples in *HowTo-SPS_install.pdf*.

6 Annex

Note: There is a problem when copying the code examples from Adobe Acrobat Reader. Please copy them from the example files laying in `xml/examples`.

6.1 CITE-plugin `n52sps:sensorConfiguration`

The following n52sps:`SensorConfiguration` example demonstrates you how you can define sensor specific information. The document structure is well defined in the `testconfig:testpluginConfiguration.xsd`. It is separated in the `testconfig:TestPluginConfiguration` and the list of `sps:InputDesciptors`. Both are embedded into the `n52sps:SensorConfiguration`.

```
<n52sps:SensorConfiguration
   xmlns="http://www.52north.org/sps/v1-00/am"
   xmlns:n52sps="http://www.52north.org/sps/v1"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   xmlns:swe="http://www.opengis.net/swe/1.0"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:gml="http://www.opengis.net/gml"
   xmlns:ows="http://www.opengis.net/ows"
   xmlns:sps="http://www.opengis.net/sps/1.0"

<testconfig:TestPluginConfiguration
   xmlns:testconfig="http://www.52north.org/sps/plugin/testconfig"
   xsi:schemaLocation="http://www.52north.org/sps/plugin/testconfig
          ./testpluginConfiguration.xsd">
   <testconfig:InstanceConfig>
      <testconfig:Phenomenon>
         urn:ogc:feature:phenomenon:AOI
      </testconfig:Phenomenon>
      <testconfig:DefaultDataServices>
         <n52sps:Service>
            <n52sps:ServiceType>SOS</n52sps:ServiceType>
               <n52sps:ServiceURL>
                  http://mars.uni-muenster.de:8080/OWS5SOS/sos
               </n52sps:ServiceURL>
            </n52sps:Service>
         </testconfig:DefaultDataServices>
      <testconfig:SensorDescription>
         <n52sps:SensorML
            xmlns:sml="http://www.opengis.net/sensorML/1.0">
            <sml:SensorML>
               <sml:member>
                  <sml:ContactList>
                     <sml:ResponsibleParty>
                        52°North
                     </sml:ResponsibleParty>
                  <sml:ResponsibleParty>
                     ifgi
                  </sml:ResponsibleParty>
                  </sml:ContactList>
               </sml:member>
            </sml:SensorML>
         </n52sps:SensorML>
      </testconfig:SensorDescription>
```

```
    <testconfig:AreaOfService>
      <ows:WGS84BoundingBox>
        <ows:LowerCorner>-90.0 -180.0</ows:LowerCorner>
        <ows:UpperCorner>90.0 180.0</ows:UpperCorner>
      </ows:WGS84BoundingBox>
    </testconfig:AreaOfService>
    <testconfig:TaskAccess>
      <testconfig:DataServices>
        <n52sps:TaskService>
          <n52sps:ServiceType>SOS</n52sps:ServiceType>
          <n52sps:ServiceURL>
            http://mars.uni-muenster.de:8080/OWS5SOS/sos
          </n52sps:ServiceURL>
        </n52sps:TaskService>
      </testconfig:DataServices>
    </testconfig:TaskAccess>
  </testconfig:InstanceConfig>

  <testconfig:InputDescription>
    <sps:InputDescriptor parameterID="measurementFrequency"
      use="optional" updateable="true">
      <sps:definition>
        <sps:commonData>
          <swe:Quantity>
            <swe:uom code="Hz"
            xlink:href="urn:x-ogc:def:uom:OGC:Hz" />
          </swe:Quantity>
        </sps:commonData>
      </sps:definition>
    </sps:InputDescriptor>
    <sps:InputDescriptor parameterID="measurementLocation"
      use="required" updateable="false">
      <sps:definition>
        <sps:commonData>
          <swe:Position
      referenceFrame="urn:ogc:def:crs:EPSG:6.14:4326"
      definition="urn:ogc:def:phenomenon:measurement_location">
          <swe:location>
            <swe:Vector>
              <swe:coordinate
                name="Geodetic latitude">
                <swe:Quantity
                  axisID="Lat">
                  <swe:uom code="deg" />
                  <swe:constraint>
                  <swe:AllowedValues>
                    <swe:interval>
                      -90 90
                    </swe:interval>
                  </swe:AllowedValues>
                  </swe:constraint>
                </swe:Quantity>
              </swe:coordinate>
              <swe:coordinate
                name="Geodetic longitude">
                <swe:Quantity
                  axisID="Long">
```

```
                    <swe:uom code="deg" />
                    <swe:constraint>
                    <swe:AllowedValues>
                      <swe:interval>
                        -180 180
                      </swe:interval>
                    </swe:AllowedValues>
                    </swe:constraint>
                  </swe:Quantity>
                </swe:coordinate>
              </swe:Vector>
            </swe:location>
            </swe:Position>
          </sps:commonData>
        </sps:definition>
      </sps:InputDescriptor>
      <sps:InputDescriptor parameterID="measurementCount"
        use="optional" updateable="false">
        <sps:definition>
          <sps:commonData>
            <swe:Count>
              <swe:constraint>
                <swe:AllowedValues>
                  <swe:max>10</swe:max>
                </swe:AllowedValues>
              </swe:constraint>
            </swe:Count>
          </sps:commonData>
        </sps:definition>
      </sps:InputDescriptor>
      <sps:InputDescriptor parameterID="measurementPurpose"
        use="optional" updateable="false">
        <sps:definition>
          <sps:commonData>
            <swe:Text />
          </sps:commonData>
        </sps:definition>
      </sps:InputDescriptor>
      <sps:InputDescriptor parameterID="measurementPriority"
        use="required" updateable="false">
        <sps:definition>
          <sps:commonData>
            <swe:Category>
              <swe:constraint>
                <swe:AllowedTokens>
                  <swe:valueList>
                    low medium high
                  </swe:valueList>
                </swe:AllowedTokens>
              </swe:constraint>
            </swe:Category>
          </sps:commonData>
        </sps:definition>
      </sps:InputDescriptor>
      <sps:InputDescriptor parameterID="shallMeasure"
        use="required" updateable="true">
        <sps:definition>
```

```
          <sps:commonData>
            <swe:Boolean />
          </sps:commonData>
        </sps:definition>
      </sps:InputDescriptor>
      <sps:InputDescriptor parameterID="maxMissionDuration"
        use="optional" updateable="false">
        <sps:definition>
          <sps:commonData>
            <swe:Time>
              <swe:constraint>
                <swe:AllowedTimes>
                  <swe:interval>
                    1990-01-01 2009-01-01
                  </swe:interval>
                </swe:AllowedTimes>
              </swe:constraint>
            </swe:Time>
          </sps:commonData>
        </sps:definition>
      </sps:InputDescriptor>
    </testconfig:InputDescription>
</testconfig:TestPluginConfiguration>

</n52sps:SensorConfiguration>
```

The input descriptors define the sensors parameters and its constraints (if there are some) with the datamodel that SweCommon describes. You can find the schema files of SweCommon at http://schemas.opengis.net/.