RFC0077 - Diagnostic

Confidential, Draft

40 Pages

Abstract

Defines a proposal to perform diagnostic within an OSGi Service Platform.

0 Document Information

0.1 Table of Contents

All Page Within This Box

0.2 Status

This document specifies a technical solution to perform diagnostic for the Open Services Gateway Initiative, and requests discussion and suggestions for improvements. Distribution of this document is unlimited within OSGi.

0.3 Acknowledgement

0.4 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

0.5 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
1.0	Feb 19, 2004	Olivier Pavé. Document creation

All Page Within This Box

1.1	Mar 16, 2004	Integration of comments from Cologne meeting: • Change Device interface name to Diagnosable interface; • Change DeviceInvocationException class name to DiagInvocationException class name; • Change data type from Byre to Int of data type attribute in Variable class; • Change variableChanged method parameters from Variable Listener interface; • Remove VariableEvent class; • Update Open Issues list with all remaining remarks
1.2	Feb 9, 2005	• Remove of metatype definitions replaced by a link to RFC 69 • Replace Diagnosable interface to DiagnosableControlUnit interface to establish a link to RFC 82 – ControlUnit
1.3	Mar 2, 2005	Incorporation of minor remarks and typos.
1.4	Mar 22, 2005	• Rework technical solution introduction with additional sections on entities and class diagram. • Update API

All Page Within This Box

1 Introduction

This document describes a generic mechanism to perform diagnostic within an OSGi Service Platform. The intention was originally to focus on diagnostic in an automotive context but it seems that the same mechanism can be used in other domain. This solution is not only about in-vehicle diagnostic.

This proposal is abstract from the content of the diagnostic commands, the protocol used to invoke these commands and type of devices diagnosed.

2 Application Domain

2.1 Automotive Context

It must be understood that the OSGi Service Platform is a node on the in-vehicle network. Diagnostic commands can be applied on all the services installed in the OSGi Service Platform.

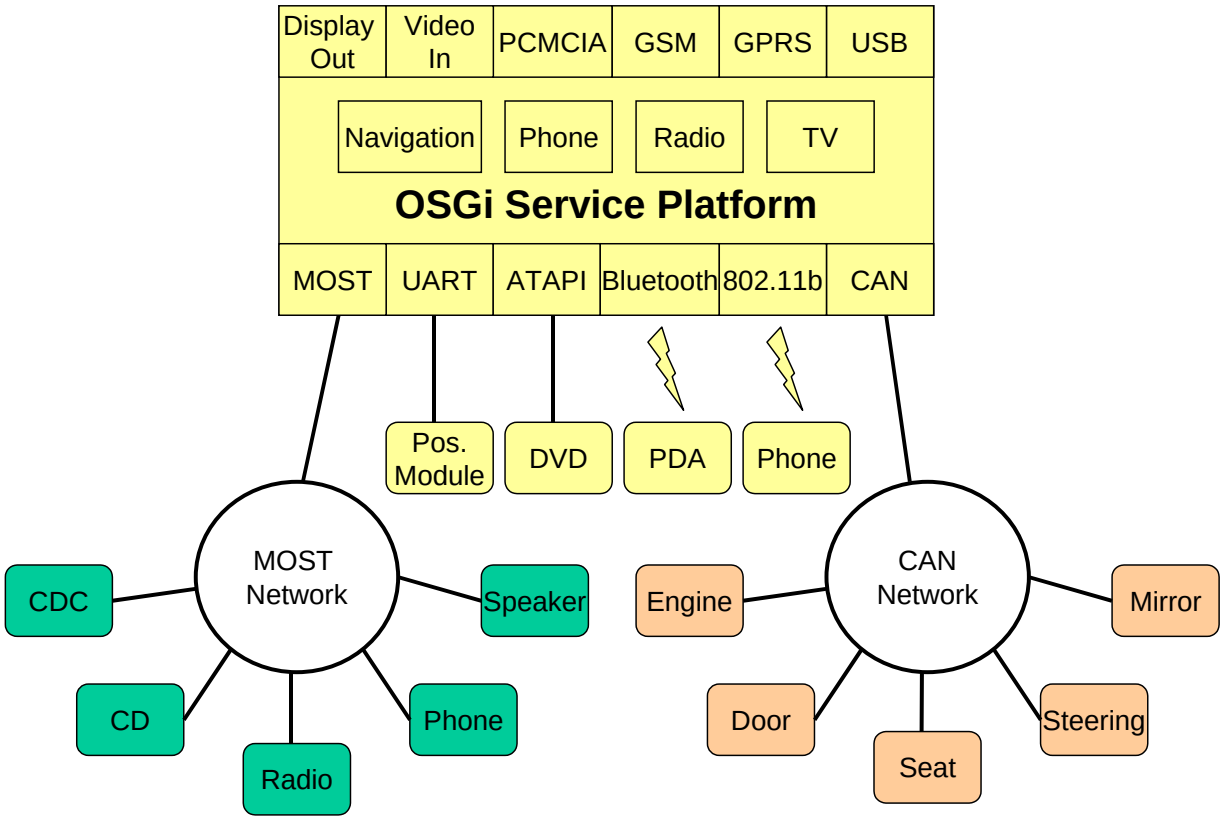


Figure 1 – Typical Vehicle Architecture

The Figure 1 presents a typical Vehicle Architecture where several devices are connected to the MOST network, the CAN network or to the OSGi Service platform directly.

The scope of this RFP is only to diagnose all software services (with maybe a device connected to it) installed on the OSGi Service Platform. All modules concerned are represented in yellow.

The other modules (in green and orange) provide diagnostic capabilities accessible via the network itself.

2.2 Entities

The following entities are used in this document:

- ✓ **Diagnostician** - A person who needs to perform a diagnostic check.
- ✓ **Service** - An OSGi service that can be connected to a device.
- ✓ **Device** - An external piece of hardware connected to the system.

2.3 Domain

The diagnostic domain is the set of commands that a diagnostician can perform on services installed on the system.

All Page Within This Box

All services have various diagnostic capabilities. We can illustrate these capabilities by using the example of an independent positioning device connected to the in-vehicle system.

The diagnostician would like to:

- ✓ **Get Device Parameter Values**
  usually a device has parameters, which may or may not be configurable. These parameters are sometimes stored on the device itself. For example, the diagnostician would like to retrieve the current parameter value and the status of the GPS antenna.

- ✓ **Change Device Parameter Values**
  when the device has configurable parameters, the diagnostician may want to change them. For example, change the DPP (Distance Per Pulse) of the Tachometer.

- ✓ **Execute a Self-test**
  some devices have a self-test feature. A test can run internally on the device and the resulting status is returned to the caller. For example, the diagnostician may want to run a self-test on the gyroscope. Some self-tests can run automatically on a device during device start-up or during runtime. If a problem is encountered, an error is logged.

- ✓ **Monitor Device Data**
  Some devices provide a continuous flow of data and a diagnostician can monitor this data. For example, a positioning device produces dead reckoning data (latitude, longitude, altitude, heading, speed, and driving direction) that is constantly monitored by a test application.

- ✓ **Collect Device Data**
  A diagnostician may want to collect device data over a period of time or at regular intervals, and store the data for future analysis. For example, on a positioning device, a diagnostician may want to know the maximum speed measured in a given period.

Note: Diagnostic command content and methods (e.g. protocols) are outside the scope of this RFP.

2.4 Standard References

- ❑ **OSGi State Management**
  The State management on an OSGi Service Platform is handled using Wire Admin Service. This provides a standard connection between state providers and consumers.

- ❑ **Keyword Protocol 2000 (ISO 14230)**
  ISO 14230 is a protocol that is used mostly on vehicles manufactured and sold in Europe. It is also commonly referred to as "Keyword Protocol 2000". In recent years, there has been increased effort on European standardization (European On Board Diagnostics or EOBD). In addition, there is limited provision in SAE J1979 for the use of certain ISO 14230 modes, so some of these vehicles may start appearing in the US in the near future. It is for the above reasons that solutions for this protocol have been developed.

All Page Within This Box

3 Problem Description

Taking into account all diagnostic capabilities defined in the application domain, there must be a standard approach for the use of diagnostic commands and for the retrieval of the resulting feedback.

Recently developed services can extend these diagnostic capabilities so the OSGi Service Platform must facilitate their use without any changes to the system.

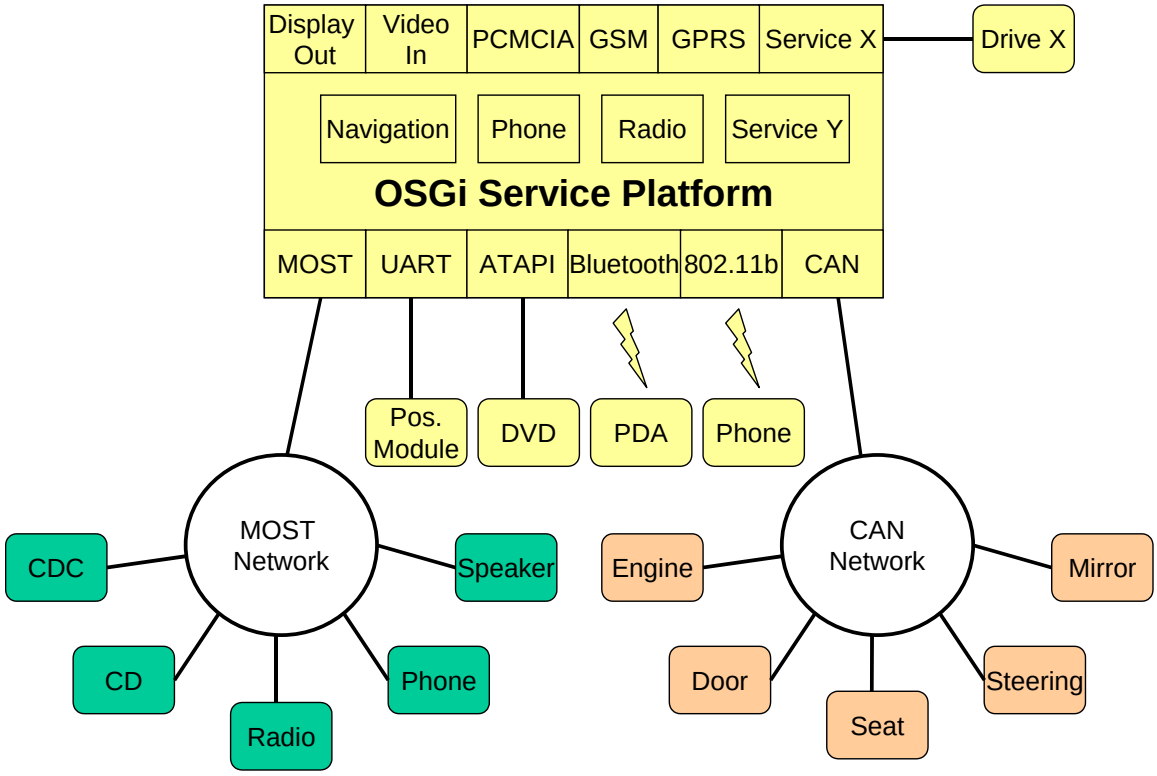


Figure 2 – Installation of two new services

In the Figure 2 two new services is installed: Service X and Service Y. The second one has a device connected to it. These two services provide new diagnostic commands and they must implement these commands in order to be easily called by a client. More, all errors and results must be properly logged in the OSGi Service Platform.

A client that needs to perform diagnostics on a service must know the extent of the diagnostic capabilities and how to use them. The framework must not be impacted.

3.1 Use Cases

Use Case Name	Diagnose a device

Feature					
Actors	*Diagnostician*	*In car user*	*Certified System Engineer*	*Other Sub-Systems*	
	X		*X*		
Description	The intent of the use case is to diagnose a device connected to the system.				
Flow of events	**Basic flow - Diagnose and adjust device parameters** The diagnostician may adjust device parameters: • When the type of the car changes. • For diagnostics purpose. 1. This use case starts when the user requests to view or change device parameters. 2. The system displays device parameter values 3. The user changes the values. 4. The system saves the changes. 5. The use case ends.				
Must have devices	A device connected to the system.				
May have devices	*N/A*				
Prior conditions	Service is running.				
Post conditions	**Basic flow:** 1. Device parameters are changed. **Alternative flows:** 1. Errors are logged and device parameters are stored.				
Alternative flow 1	**The system tests the service** 1. This use case starts when the diagnostician requests a self-test. 2. The system starts the self-test on the service 3. The system logs errors and warnings. 4. The system returns a result from the self-test to the caller				

All Page Within This Box

	5. The use case ends.
Alternative flow 2	**Diagnosis of device input data** 1. This use case starts on the diagnostician requests a record. 2. The system starts the collection of device data 3. The system shall be able to provide the following data on a continuous basis on screen. 4. The use case ends.
Alternative flow 3	**The system collects device data for a given period and store them** Specific prior-conditions: • The system is continuously supplied by electrical power. • The collection period is customizable. 1. This use case starts when the diagnostician requests to collect information. 2. The user can change the collection period. 3. The system starts the collection of data. 4. The system stores the collected data into persistent memory. 5. The use case ends.
Notes	
Future considerations	Data collection The system provides a facility to output a file with the collected device data.
Lower level Use Cases	*N/A*
Date	02/04/03
Status	Draft

All Page Within This Box

4 Requirements

Remote and Local Connection

A diagnostician should be able to perform diagnostics locally or remotely. Locally means that a test application is installed on the system and displays/saves the results from the tests locally (e.g. on the car monitor). Remotely means that the diagnostician performs diagnostics through a communication channel (MOST, CAN, TCP/IP, Wireless, …)

Change Device Parameters

A diagnostician must be able to view, change and if necessary store device parameter values.

Start Self-test on a Service

The system must allow starting (and stopping) of a self-test on a service. Self-test results can be stored on a logger and/or sent out to the test application.
Self-tests can be started explicitly by the diagnostician, or can run automatically during device (if any) start-up or during runtime.

Monitor Device Data

The system must allow the setup of a continuous flow of a defined set of data from a device. Data are sent continuously to the test application. The test application can decide whether to display it or record it or send it to be displayed remotely.

Collect Device Data

The system must allow the collection of a defined set of data from a device. The collection period must be configurable and the data stored in persistent memory, and if necessary, displayed on the test application.

Read Logged Information

The system must allow the logged information to be read during a self-test and after a system crash.

Diagnose Newly Installed Service

The system must allow the diagnosis of newly installed services without any changes to the system.

Notify about Bundle Degradation

There is no standardized way of supervising bundles except for checking that the state of the bundle is "ACTIVE". A bundle may be marked as "ACTIVE" even though it is not functioning. In this case the bundle wants to inform client that something is going and that their internal state is not really "ACTIVE". The main goal is to allow client to register to bundle specific internal states.

Separable Diagnostic Commands

Diagnostic commands are usually used on demand. For low memory devices it should possible to avoid a permanent installation of bundles containing diagnostic capabilities. These bundles could be installed when needed.

5 Technical Solution

5.1 Introduction

The technical solution proposed is based on Control Unit concept (see RFC 82).

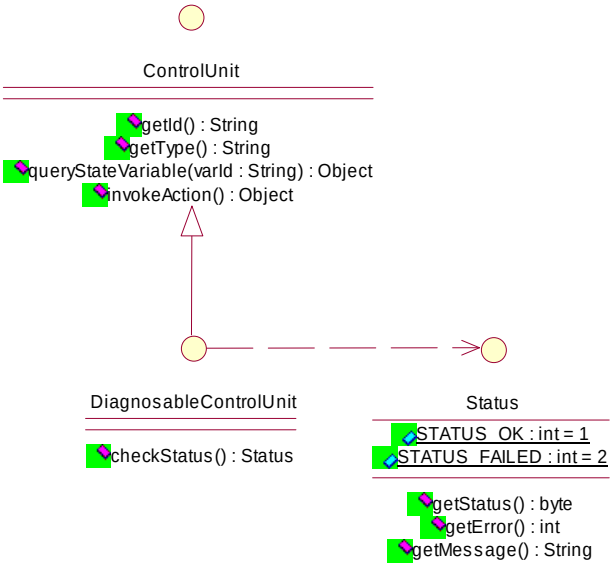


Figure 3 – Class Diagram

As a reminder, the Control Unit concept is based on the assumption that devices can be generally represented by a set of attributes (or state variables) and actions.

The Control Unit specification defines the way to invoke actions and retrieve state variable values in addition some Meta data on these units can be stored and retrieved in the Metatype service.

For a diagnostic point view, the context is the same. A diagnostician wants to either perform an action or query/listen to a value offered by a device.

The Control Unit concept can then be used to:

- Discover diagnostic capabilities of devices
- Invoke one of these diagnostic commands
- Listen to device state variable changes

The `DiagnosableControlUnit` interface extends the basic `ControlUnit` interface to add a diagnostic specific method: `checkStatus()`.

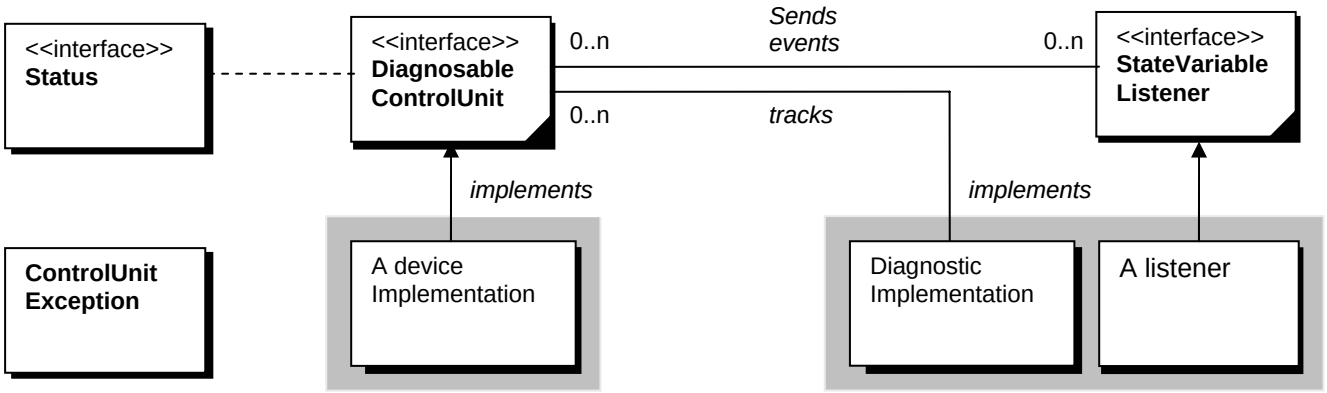
This method is called to perform a self-test on the device. It returns a simple status (OK or FAILED). It is up to the device to decide the kind of diagnostic commands that need to be performed to check if the device is correctly running or not.

All Page Within This Box

See RFC 82 for detailed information on Control Units.

5.2 Entities

- *Action* – An action that can be invoked on a ControlUnit using `invokeAction()` method. In the diagnostic context an action is a diagnostic command. An action can be described using metatype specification.
- *ControlUnit* – Representation of a device in terms of actions and state variables.
- *ControlUnitException* – An exception that can be thrown when an action invocation fails on a control unit.
- *DiagnosableControlUnit* – A control unit that provides actions and state variables relative to diagnostic.
- *StateVariable* – A variable published by a control unit. This variable must be uniquely identified. In the diagnostic context a state variable is a data that can be monitored.
- *StateVariableListener* – A listener on state variable changes. This listener has to be registered within the framework to receive control unit notification.
- *Status* – The result of `checkStatus()` method.



5.3 Publish diagnostic device capabilities

A device that would like to publish some diagnostic capabilities must implement `DiagnosableControlUnit` interface and register it within the framework. As soon as the services are registered new diagnostic capabilities are available and any test applications can use them.

To be correctly identified these services can register properties like the CU id.

In addition the device can offer some descriptions about these capabilities by implementing a metatype provider or by providing a XML file containing metatype information (see RFC 69).

All Page Within This Box

5.4 Discover diagnostic device capabilities

An application that would like to discover diagnostic capabilities will have to search in the Metatype Service using the Type of the Control Unit (see RFC 82). Metadata can be used to generate automatically a user interface. This UI will allow the user to perform diagnostic commands. It will be automatically updated according to service registration / unregistration.

5.5 Invoke diagnostic commands

An application that would like to perform diagnostic commands has to get the appropriate service from the framework using the CU id.

There are two different possibilities:

- Command are known by the test application

  In this case the test application can directly use methods offered by DiagnosableControlUnit interface to invoke diagnostic commands.

- Commands are not known by the test application

  In this case the test application will have to retrieve metadata to discover actions (if any) and the set of attributes (if any) offered by the device

Three different kinds of methods can be called:

- `InvokeAction`: to perform any diagnostic commands
- `queryStateVariable`: to retrieve a state variable value from the device
- `checkStatus`: to perform a self-check of the device.

5.6 Listen to state variable changes

Some test applications want to monitor state variable changes of a device. In this case the application will have to implement a `StateVariableListener` service and register it within the framework.

While registering the service the application can add properties that will perform a filter on event received. The filter is specified as an LDAP syntax expression over control unit Id, control unit type and state variable Id.

All Page Within This Box

5.7 API

5.7.1 Interface DiagnosableControlUnit

All Superinterfaces:

ControlUnit

public interface **DiagnosableControlUnit**
extends ControlUnit

Describes a device in terms of actions that can be invoked on the device and variable that can retrieved from a device.

Version:

$Revision: 1.1 $

Method Summary

Status	**checkStatus**() Performs a complete diagnostic on the device.

Methods inherited from interface org.osgi.service.cu.ControlUnit
getId, getType, invokeAction, queryStateVariable

Method Detail

checkStatus

```
public Status checkStatus()
                    throws ControlUnitException
```

Performs a complete diagnostic on the device. It is up to the device to decide what are the actions that must be performed to check if the device is correctly running. The result is given by the status.

Returns:

The status of the method

Throws:

ControlUnitException - if error prevents the execution of the action. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.

All Page Within This Box

5.7.2 Interface Status

public interface **Status**

Indicates the return status of a selt-test performed over a Device by calling method checkStatus(). The status value can be: STATUS_OK or STATUS_FAILED. In case of STATUS_FAILED, the getError() method returns the error that occured.

Version:
$Revision: 1.1 $

Field Summary

`static int`	**`STATUS_FAILED`** Indicates that the method was not successfully performed.
`static int`	**`STATUS_OK`** Indicates that the method was successfully performed.

Method Summary

`int`	**`getError`**`()` Returns the error code if the status is STATUS_FAILED.
`java.lang.String`	**`getMessage`**`()` Returns the additional information on the status.
`byte`	**`getStatus`**`()` Returns the method status after invocation.

Field Detail

STATUS_OK

`public static final int` **`STATUS_OK`**

Indicates that the method was successfully performed.
See Also:
Constant Field Values

STATUS_FAILED

`public static final int` **`STATUS_FAILED`**

Indicates that the method was not successfully performed.
See Also:
Constant Field Values

Method Detail

getStatus

All Page Within This Box

`public byte` **`getStatus`**`()`

Returns the method status after invocation. The status can be one of the following values: STATUS_OK or STATUS_FAILED.

Returns:

The method status after invocation

---

getError

`public int` **`getError`**`()`

Returns the error code if the status is STATUS_FAILED.

Returns:

The error code if the status is STATUS_FAILED, -1 if there is no error code.

---

getMessage

`public java.lang.String` **`getMessage`**`()`

Returns the additional information on the status. The error message can contain the actionID that was not successful.

Returns:

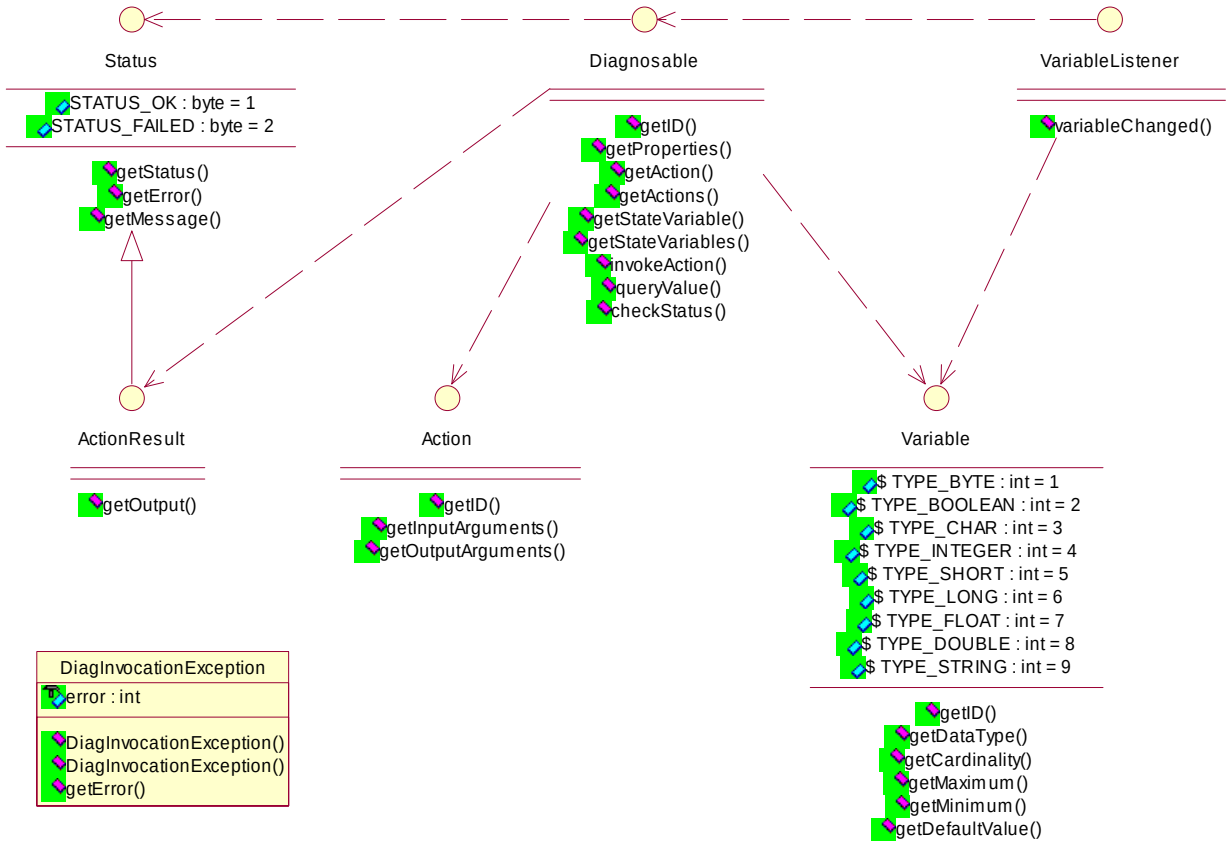
The error message or null if there is no message.

6 Considered Alternatives

6.1 Diagnosable interface

The solution is based on a generic mechanism to describe actions that can be invoked on a device and state variable that can be queried from a device.

6.1.1 Class Diagram



The `Diagnosable` interface is the main interface where you can:

- get all diagnostic actions
- get all state variables
- invoke a diagnostic action

All Page Within This Box

- query a value
- check service status

The `Action` interface defines the content of an action that can be invoked on a service. It contains an ID to uniquely identify an action, the input arguments and the output arguments.

The `Variable` interface defines the content of a variable that can be received from a service. It contains an ID to uniquely identify a variable and some other information that defined a variable. This class is also used to describe the arguments of an `Action`.

A client can implement a `VariableListener` and register it to the framework in order to listen to `Variable` value changes. For each changes the listener receives a reference to the diagnosable interface, a reference to the variable that has changed and the new variable value.

6.1.2 API

All the following classes are defined in `org.osgi.nursery.diagnostic`.

6.1.2.1 Interface Action

public interface **Action**

Contains the definition of an Action. An action defined a command that a client can invoke on a device by using inokeAction() method.

Method Summary

`java.lang.String`	**`getID`**`()` Returns the ID of the action.
`Variable[]`	**`getInputArguments`**`()` Returns the list of input arguments.
`Variable[]`	**`getOutputArguments`**`()` Returns the list of output arguments.

Method Detail

getID

`public java.lang.String` **`getID`**`()`

Returns the ID of the action.

Returns:

The ID of the action.

getInputArguments

`public Variable[]` **`getInputArguments`**`()`

Returns the list of input arguments. These arguments are defined by using Variable objects.

Returns:

The list of input arguments or null if not defined.

getOutputArguments

`public Variable[]` **`getOutputArguments`**`()`

Returns the list of output arguments. These arguments are defined by using Variable objects.

Returns:

The list of output arguments or null if not defined.

All Page Within This Box

6.1.2.2 Interface ActionResult

All Superinterfaces:

Status

---

public interface **ActionResult**
extends Status

Indicates the return status of an Actionthe method. The status value can be: STATUS_OK or STATUS_FAILED In case of STATUS_FAILED the getError method returns the error that occured.

Field Summary

Fields inherited from interface org.osgi.nursery.diagnostic.Status
STATUS_FAILED, STATUS_OK

Method Summary

java.lang.Object[]	**getOutput**() Returns the output of the method.

Methods inherited from interface org.osgi.nursery.diagnostic.Status
getError, getMessage, getStatus

Method Detail

getOutput

public java.lang.Object[] **getOutput**()

Returns the output of the method. This output is compatible to arguments described in getOutputArguments() method.

Returns:

An array of output argument values

All Page Within This Box

6.1.2.3 Interface Diagnosable

public interface **Diagnosable**

Decsribes a device in terms of actions that can be invoked on the device and variable that can retrieved from a device.

Method Summary

Status	**checkStatus**() Performs a complete diagnostic on the device.
Action	**getAction**(java.lang.String actionID) Returns the Action object that has the given ID.
Action[]	**getActions**() Returns the list of Action object that we can invoked on this device.
java.lang.String	**getID**() Returns the ID of the device.
java.util.Dictionary	**getProperties**() Returns some properties (if any) of the device.
Variable	**getStateVariable**(java.lang.String varID) Returns the Variable object related to a given ID.
Variable[]	**getStateVariables**() Returns the list of Variable objects that this device can retrieved.
ActionResult	**invokeAction**(java.lang.String actionID, java.lang.Object[] args) Invokes an Action on a device.
java.lang.Object	**queryValue**(java.lang.String varID) Queries the value of the variable expressed in varID.

Method Detail

getAction

public Action **getAction**(java.lang.String actionID)

Returns the Action object that has the given ID.

Returns:

The action object or null if not found.

getActions

public Action[] **getActions**()

Returns the list of Action object that we can invoked on this device.

Returns:

The list of Action objects or null if empty.

getStateVariable

All Page Within This Box

`public Variable getStateVariable(java.lang.String varID)`

Returns the Variable object related to a given ID.

Parameters:

`varID` - The ID of the Variable object searched.

Returns:

The Variable object or null if not found.

---

getStateVariables

`public Variable[] getStateVariables()`

Returns the list of Variable objects that this device can retrieved.

Returns:

The lis of Variable objects or null if empty.

---

invokeAction

```
public ActionResult invokeAction(java.lang.String actionID,
                                 java.lang.Object[] args)
                          throws DiagInvocationException
```

Invokes an Action on a device. The list of value for the input arguments must be in the same order as defined in Action.InputArguments() method. The returned values are in the same order as defined in Action.outputArguments() method.

Parameters:

`actionID` - The ID of the action invoked

`args` - The list of input argument

Returns:

The list of Variable objects or null if empty.

Throws:

`java.lang.IllegalArgumentException` - if at least one argument value contained in args do not have the right data type.

`org.osgi.nursery.diagnostic.DeviceInvocationException` - if the method cannot be invoked on the device.

`DiagInvocationException`

---

queryValue

```
public java.lang.Object queryValue(java.lang.String varID)
                            throws DiagInvocationException
```

Queries the value of the variable expressed in varID.

Parameters:

`varID` - The ID of the requested variable value.

Returns:

The value of the requested variable or null if not found.

Throws:

`org.osgi.nursery.diagnostic.DeviceInvocationException` - if the method cannot be invoked on the device.

`DiagInvocationException`

---

getID

`public java.lang.String getID()`

Returns the ID of the device.

Returns:

The ID value of the device.

---

getProperties

`public java.util.Dictionary` **`getProperties`**`()`

Returns some properties (if any) of the device. These properties are contained in a dictionary and are completely

Returns:

The list of properties or null if not defined.

---

checkStatus

`public` Status **`checkStatus`**`()`
`throws` DiagInvocationException

Performs a complete diagnostic on the device. It is up to the device to decide what are the actions that must be performed to check if the device is correctly running. The result is given by the status.

Returns:

The status of the method

Throws:

`org.osgi.nursery.diagnostic.DeviceInvocationException` - if the method cannot be invoked on the device.

DiagInvocationException

6.1.2.4 Class DiagInvocationException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--org.osgi.nursery.diagnostic.DiagInvocationException
```

All Implemented Interfaces:

java.io.Serializable

---

public class **DiagInvocationException**
extends java.lang.Exception

Event sent to a VariableListener when a variable value has changed.

See Also:

Serialized Form

Constructor Summary

DiagInvocationException(int error) Creates a new object of DeviceInvocationException class.
DiagInvocationException(java.lang.String message, int error) Creates a new object of DeviceInvocationException class.

Method Summary

int	**getError**() Returns the error code.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

DiagInvocationException

```
public DiagInvocationException(java.lang.String message,
                               int error)
```

All Page Within This Box

Creates a new object of DeviceInvocationException class. This class contains message and error code in case of a device invocation failure.

Parameters:

`message` - The error message

`error` - The error code

DiagInvocationException

`public` **`DiagInvocationException`**`(int error)`

Creates a new object of DeviceInvocationException class. This class contains an error code in case of a device invocation failure.

Parameters:

`error` - The error code

Method Detail

getError

`public int` **`getError`**`()`

Returns the error code.

Returns:

The error code value.

6.1.2.5 Interface Status

All Known Subinterfaces:

ActionResult

---

public interface **Status**

Indicates the return status of a method called on a Device. The status value can be: STATUS_OK or STATUS_FAILED In case of STATUS_FAILED the getError method returns the error that occured.

Field Summary

`static byte`	**`STATUS_FAILED`** Indicates that the method was not successfully performed.
`static byte`	**`STATUS_OK`** Indicates that the method was successfully performed.

Method Summary

`int`	**`getError`**`()` Returns the error code if the status is STATUS_FAILED.
`java.lang.String`	**`getMessage`**`()` Returns the additional information on the status.
`byte`	**`getStatus`**`()` Returns the method status after invocation.

Field Detail

STATUS_OK

`public static final byte` **`STATUS_OK`**

Indicates that the method was successfully performed.

See Also:

Constant Field Values

STATUS_FAILED

`public static final byte` **`STATUS_FAILED`**

Indicates that the method was not successfully performed.

See Also:

Constant Field Values

Method Detail

getStatus

`public byte` **`getStatus`**`()`

All Page Within This Box

Returns the method status after invocation. The status can be one of the following values: STATUS_OK or STATUS_FAILED.

Returns:

The method status after invocation

---

getError

`public int` **`getError`**`()`

Returns the error code if the status is STATUS_FAILED.

Returns:

The error code if the status is STATUS_FAILED, -1 if there is no error code.

---

getMessage

`public java.lang.String` **`getMessage`**`()`

Returns the additional information on the status. The error message can contain the actionID that was unsuccessful.

Returns:

The error message or null if there is no message.

6.1.2.6 Interface Variable

public interface **Variable**

Defines the content of a Variable of a Device. This definition is used by a client to discover and use Variable from a Device. A value of a Variable can be retrieved by using queryValue() method of Device interface or by registering a VariableListener service.

Field Summary

static int	**TYPE_BOOLEAN** Type for a Boolean object.
static int	**TYPE_BYTE** Type for a Byte object.
static int	**TYPE_CHAR** Type for a Character object.
static int	**TYPE_DOUBLE** Type for a Double object.
static int	**TYPE_FLOAT** Type for a Float object.
static int	**TYPE_INTEGER** Type for a Integer object.
static int	**TYPE_LONG** Type for a Long object.
static int	**TYPE_SHORT** Type for a Short object.
static int	**TYPE_STRING** Type for a String object.

Method Summary

int	**getCardinality**() Returns the variable cardinality.
int	**getDataType**() Returns the data type of the variable as defined by constants.
java.lang.Object	**getDefaultValue**() Returns the default value if defined.
java.lang.String	**getID**() Returns the ID value of the variable.
java.lang.Number	**getMaximum**() Returns the maximum value if defined.
java.lang.Number	**getMinimum**() Returns the minimum value if defined.

All Page Within This Box

Field Detail

TYPE_BYTE

`public static final int` **`TYPE_BYTE`**

Type for a Byte object. The type is byte[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_BOOLEAN

`public static final int` **`TYPE_BOOLEAN`**

Type for a Boolean object. The type is boolean[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_CHAR

`public static final int` **`TYPE_CHAR`**

Type for a Character object. The type is char[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_INTEGER

`public static final int` **`TYPE_INTEGER`**

Type for a Integer object. The type is integer[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_SHORT

`public static final int` **`TYPE_SHORT`**

Type for a Short object. The type is short[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_LONG

`public static final int` **`TYPE_LONG`**

Type for a Long object. The type is long[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_FLOAT

`public static final int` **`TYPE_FLOAT`**

Type for a Float object. The type is float[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_DOUBLE

`public static final int` **`TYPE_DOUBLE`**

Type for a Double object. The type is double[] if the cardinality is greater than 1.

See Also:

Constant Field Values

TYPE_STRING

`public static final int` **`TYPE_STRING`**

Type for a String object.

See Also:

Constant Field Values

Method Detail

getID

`public java.lang.String` **`getID`**`()`

Returns the ID value of the variable.

Returns:

The ID value of the variable.

---

getDataType

`public int` **`getDataType`**`()`

Returns the data type of the variable as defined by constants. The type is one of the following value: TYPE_BYTE, TYPE_BOOLEAN, TYPE_CHAR, TYPE_INTEGER, TYPE_SHORT, TYPE_LONG, TYPE_FLOAT, TYPE_DOUBLE, TYPE_STRING

Returns:

The value of the data type.

---

getCardinality

`public int` **`getCardinality`**`()`

Returns the variable cardinality. If the cardinality is greater than 1 it means that the type is an array of the primitif type returned by getDataType() method.

Returns:

The variable cardinailty.

---

getMaximum

`public java.lang.Number` **`getMaximum`**`()`

Returns the maximum value if defined. Maximum value can only be defined for numeric types.

Returns:

The maximum value or null if not defined.

---

getMinimum

`public java.lang.Number` **`getMinimum`**`()`

Returns the minimum value if defined. Minimum value can only be defined for numeric types.

Returns:

The minimum value or null if not defined.

---

getDefaultValue

`public java.lang.Object` **`getDefaultValue`**`()`

Returns the default value if defined.

Returns:

The default value or null if not defined.

6.1.2.7 Interface VariableListener

public interface **VariableListener**

Interface of listener to device variable value changes. This interface must be implemented and registered to the framework in order to recieve events. If a listener wants to filter these events he has to register the property VARIABLE_FILTER during listener registration. If the filter syntax is incorrect the filter is ignored.

Field Summary

`static java.lang.String`	**`VARIABLE_FILTER`** Property name used to indicate a filter.

Method Summary

`void`	**`variableChanged`**`(Diagnosable diag, Variable var, java.lang.Object value)` This method is called to inform a listener that a variable has changed.

Field Detail

VARIABLE_FILTER

```
public static final java.lang.String VARIABLE_FILTER
```

Property name used to indicate a filter. The value is a String indicating the filter. The valid constants used for the filter definition are: Device.ID - The ID of a specific device to listen for events Variable.ID - The ID of the variable to listen for events

See Also:

Constant Field Values

Method Detail

variableChanged

```
public void variableChanged(Diagnosable diag,
                            Variable var,
                            java.lang.Object value)
```

This method is called to inform a listener that a variable has changed.

Parameters:

`diag` - The diagnosable interface where this Variable is coming from

`var` - Variable that has changed

`value` - The new value of this variable

All Page Within This Box

6.1.3 Advantages

- It is extendable to handle all diagnostic commands.
- There is no direct dependency (Java package dependency) between a service that offers diagnostic commands and the client that uses them.
- It is a very abstract solution that can be applied to other domain especially for user interface issues.

6.1.4 Drawbacks

- The semantics and syntax of diagnostic commands are more-less hidden behind device interface.
- The source code of the test application is not verified at compile time because there is no direct dependency. Errors and misuses will be found during runtime when these commands will be invoked.
- Complex object are handled using byte[].

6.2 Wire Admin

6.2.1 Solution Description

The solution is based on the Wire Admin Service defined in the OSGi Service Platform Release 3.0. The main goal of this service is to connect data producers to data consumers. A service that offers diagnostic commands can be seen as data producer and test application can be seen as a data consumer.

When a test application needs to collect or monitor data then it needs to connect to another kind of producer (e.g. producer of Position).

There are two solutions to implement diagnostic commands:

- Use of a Composite Producer

  The whole diagnostic commands are represented by a single Composite Producer. The producer provides data as an array of Envelope. There are problems to adapt different frequency of production because all data do not have the same frequency. The same problem occurs with filters because the data produced are not filtered in the same way by the consumer

- Use of a set of Producer

  In this configuration the implementation of the diagnostic commands is more complex because each single data produced is represented as a Producer. But this solution is more flexible because the update frequency can be different and the data filtered can be done per data by the consumer.

6.2.2 Concrete example

As an implementation of this solution we can implement the diagnostic commands of a HIP module that delivers dead reckoning positions.

The HIP module produces data. It can also consume some data in order to calibrate internal algorithms.

Data produced are:

- Position
  It is the dead reckoning position. The update frequency is 5 Hz.
- Software Version
- Hardware Version
- GPS Date and Time
- GPS
  Gives information about GPS and all GPS channels used.
- Gyroscope
- Tachometer
- Direction Switch

Data consumed are:

All Page Within This Box

- Position

  A position can be set at the initialization phase if known. Other positions can be delivered by the map matcher and consumed by the HIP module in order to improve accuracy.

- Gyroscope

  Data used for gyro calibration.

- Tachometer

  Data used for tachometer calibration.

- Direction Switch

  Data used for direction switch calibration.

There are three possibilities to implement this example for the producer side:

1. One global producer

   This solution has the lower cost because only one producer must be implemented but it is less clear how to use it.

2. Several producers to group

   This solution seems to be the most appropriate solutions. We have to group data according to their relationship and frequency. We can identify four different producers: one for Position, one for general information (software version, hardware version) one for GPS information and one for additional information (Gyro, Tacho and Direction Switch)

3. One producer per data

   It is the most flexible solution but with the higher cost. You have to implement one producer per data that is to say in our case eight producers.

6.2.3 Security

As mentioned by the Wire Admin Service specification the security can be controlled in several levels.

At service level, ServicePermission[REGISTER,Producer|Consumer] should not be restricted. ServicePermission[GET,Producer|Consumer] must be limited to trusted bundles. It means that diagnostic command producers should not be limited but the diagnostic consumer (Test application) can be restricted using GET ServicePermission.

At Wire Admin level, all sensitive methods to create, update and delete wires could be limited using the GET ServicePermission on the Wire Admin Service.

At the wire level, the use of a consumer or producer can be limited using WirePermission[CONSUME|PRODUCE, Scope].

6.2.4 Advantages

- This solution already exists in the OSGi Service Platform specification.
- It is the "natural" solution because the Wire Admin Service has been designed to act as a bus image (e.g. MOST).
- All security issues are already solved because wire creations can be secured and there is a notion of scope.

All Page Within This Box

- It is easy to delete wires, uninstall consumer (test application), and uninstall producers (diagnostic commands). This means that there is no overhead when diagnostic commands are not used.

6.2.5 Drawbacks

- A peer-to-peer approach is not possible because a consumer can be connected to several producer and vice-versa.

- There is a dependency between producer and consumer on the data type exchanged. This means that the diagnostic commands must be declared in advance, the Java reflection is not used in this case.

- According to the solution diagnostic commands must be split into Producers and Consumers. This work must be done with particular attention because bad choices can lead to a solution that can be hard to use.

6.2.6 Conclusion

The Wire Admin solution is not fully suitable to solve diagnostic issues.

The producing/consuming of diagnostic data can easily been done using Wire Admin but the execution and result retrieving of diagnostic commands can not be done.

When a diagnostic is performed the tester wants to connect to a well defined device (peer to peer connection) and perform commands on this device. In the Wire Admin model a producer can be delivered data to several consumers and a consumer can retrieve data from several producers.

6.3 Diagnostic Adapters

6.3.1 Technical Solution

A Diagnostic Adapter is used as a utility class to perform diagnostic commands. As soon as the Diagnostic Adapter is linked to a Diagnostic Service a command can be performed by using declareDiagnostic() method to set the method name and setParameter() method to set arguments of this method. The execute() method is used to start command execution.

This solution is using Java reflection to find and invoke the right method.

6.3.2 Advantages

- There is no direct dependency (Java package dependency) between a service that offers diagnostic commands and the client that uses them.

6.3.3 Drawbacks

- The client source code is not easily readable.
- Invocation errors are found during runtime and not at compile time.
- The use of Java reflection can lead to an overhead on time execution.

6.3.4 Conclusion

It is another good alternative but a direct use of an interface is preferred to the use of Java reflection.

7 Security Considerations

Using diagnostic commands must be limited to a well known set of bundle because these commands have a direct effect on the system.

It is therefore recommended that ServicePermission[REGISTER|GET,DiagnosableControlUnit] be used sparingly and only for bundles that are trusted.

8 Document Support

8.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. OSGi Service Platform specification, Release 3.0, March 2003

[4]. RFP 37 – Diagnostic

[5]. RFP 47 – Control Unit

[6]. RFC 82 – Control Unit

8.2 Author's Address

Name	Olivier Pavé
Company	Siemens VDO Automotive
Address	Batiment Alpha 80, route des Lucioles – BP 305 06906 Sophia Antipolis Cedex
Voice	+33 (0)4 92 38 11 29
e-mail	olivier.pave@siemens.com

All Page Within This Box

8.3 Acronyms and Abbreviations

HIP

Host Independent Positioning. A separated module that delivers dead reckoning positions.

8.4 End of Document