OSGi Service Platform, Vehicle Specification Release 4

OSGi Service Platform Vehicle Specification

The OSGi Alliance

Release 4
August 2005

IOS Press
Ohmsha

Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

Table Of Contents

ISBN 1 58603 311 5 (IOS Press)
ISBN 4-274-90559-4 (Ohmsha)
Library of Congress Control Number: 2003107481

Publisher

IOS Press
Nieuwe Hemweg 6B
1013 BG Amsterdam
The Netherlands
fax: +31 20 620 3419
e-mail: order@iospress.nl

Distributor in the UK and Ireland
IOS Press/Lavis Marketing
73 Lime Walk
Headington
Oxford OX3 7AD
England
fax: +44 1865 75 0079

Distributor in the USA and Canada
IOS Press, Inc.
5795-G Burke Centre Parkway
Burke, VA 22015
USA
fax: +1 703 323 3668
e-mail: iosbooks@iospress.com

Distributor in Germany, Austria and Switzerland
IOS Press/LSL.de
Gerichtsweg 28
D-04103 Leipzig
Germany
fax: +49 341 995 4255

Distributor in Japan

Ohmsha, Ltd.
3-1 Kanda Nishiki-cho
Chiyoda-ku, Tokyo 101-8460
Japan
fax: +81 3 3233 2426

Feedback

This specification can be downloaded from the OSGi web site:
http:// www.osgi.org.

Comments about this specification can be mailed to:
speccomments@mail.osgi.org

OSGi Member Companies

4DHomeNet, Inc.	Acunia
Alpine Electronics Europe Gmbh	AMI-C
Atinav Inc.	BellSouth Telecommunications, Inc.
BMW	Bombardier Transportation
Cablevision Systems	Coactive Networks
Connected Systems, Inc.	Deutsche Telekom
Easenergy, Inc.	Echelon Corporation
Electricite de France (EDF)	Elisa Communications Corporation
Ericsson	Espial Group, Inc.
ETRI	France Telecom
Gatespace AB	Hewlett-Packard
IBM Corporation	ITP AS
Jentro AG	KDD R&D Laboratories Inc.
Legend Computer System Ltd.	Lucent Technologies
Metavector Technologies	Mitsubishi Electric Corporation
Motorola, Inc.	NTT
Object XP AG	On Technology UK, Ltd
Oracle Corporation	P&S Datacom Corporation
Panasonic	Patriot Scientific Corp. (PTSC)
Philips	ProSyst Software AG
Robert Bosch Gmbh	Samsung Electronics Co., LTD
Schneider Electric SA	Siemens VDO Automotive
Sharp Corporation	Sonera Corporation
Sprint Communications Company, L.P.	Sony Corporation
Sun Microsystems	TAC AB
Telcordia Technologies	Telefonica I+D
Telia Research	Texas Instruments, Inc.
Toshiba Corporation	Verizon
Whirlpool Corporation	Wind River Systems

OSGi Board and Officers

Foreword

John Barr, *President OSGi*

1 Introduction

The Open Services Gateway Initiative (OSGi™) was founded in March 1999. Its mission is to create open specifications for the network delivery of managed services to local networks and devices. The OSGi organization is the leading standard for next-generation Internet services to homes, cars, small offices, and other environments.

The OSGi service platform specification delivers an open, common architecture for service providers, developers, software vendors, gateway operators and equipment vendors to develop, deploy and manage services in a coordinated fashion. It enables an entirely new category of smart devices due to its flexible and managed deployment of services. The primary targets for the OSGi specifications are set top boxes, service gateways, cable modems, consumer electronics, PCs, industrial computers, cars and more. These devices that implement the OSGi specifications will enable service providers like telcos, cable operators, utilities, and others to deliver differentiated and valuable services over their networks.

This is the third release of the OSGi service platform specification developed by representatives from OSGi member companies. The OSGi Service Platform Release 3 mostly extends the existing APIs into new areas. The few modifications to existing APIs are backward compatible so that applications for previous releases should run unmodified on release 3 Frameworks. The built-in version management mechanisms allow bundles written for the new release to adapt to the old Framework implementations, if necessary.

1.1 Sections

1.2 What is New

1.3 Reader Level

This specification is written for the following audiences:

- Application developers
- Framework and system service developers (system developers)
- Architects

This specification assumes that the reader has at least one year of practical experience in writing Java programs. Experience with embedded systems and server environments is a plus. Application developers must be aware that the OSGi environment is significantly more dynamic than traditional desktop or server environments.

System developers require a *very* deep understanding of Java. At least three years of Java coding experience in a system environment is recommended. A Framework implementation will use areas of Java that are not normally encountered in traditional applications. Detailed understanding is required of class loaders, garbage collection, Java 2 security, and Java native library loading.

Architects should focus on the introduction of each subject. This introduction contains a general overview of the subject, the requirements that influenced its design, and a short description of its operation as well as the entities that are used. The introductory sections require knowledge of Java concepts like classes and interfaces, but should not require coding experience.

Most of these specifications are equally applicable to application developers and system developers.

1.4 Conventions and Terms

1.4.1 Typography

A fixed width, non-serif typeface (`sample`) indicates the term is a Java package, class, interface, or member name. Text written in this typeface is always related to coding.

Emphasis (*sample*) is used the first time an important concept is introduced.

When an example contains a line that must be broken over multiple lines, the « character is used. Spaces must be ignored in this case. For example:

```
http://www.acme.com/sp/ «
file?abc=12
```

is equivalent to:

```
http://www.acme.com/sp/file?abc=12
```

In many cases in these specifications, a syntax must be described. This syntax is based on the following symbols:

```
*            Repetition of the previous element zero or
             more times, e.g. ( ',' list ) *
+            Repetition one or more times
?            Previous element is optional
( ... )      Grouping
'...'        Literal
|            Or
[...]        Set (one of)
..           list, e.g. 1..5 is the list 1 2 3 4 5
<...>        Externally defined token
digit        ::= [0..9]
alpha        ::= [a..zA..Z]
token        ::= alpha(alpha|digit|'_'|'-')*
quoted-string::= '"' ... '"'
jar-path     ::= file['/'file]
file         A valid file name in a zip file which
```

```
has no restrictions except that it may not
contain a '/'.
```

Spaces are ignored unless specifically noted.

1.4.2 Object Oriented Terminology

Concepts like classes, interfaces, objects, and services are distinct but subtly different. For example, "LogService" could mean an instance of the class `LogService`, could refer to the class `LogService`, or could indicate the functionality of the overall Log Service. Experts usually understand the meaning from the context, but this understanding requires mental effort. To highlight these subtle differences, the following conventions are used.

When the class is intended, its name is spelled exactly as in the Java source code and displayed in a fixed width typeface: for example the "`HttpService` class", "a method in `HttpContext`" or "a `javax.servlet.Servlet` object". A class name is fully qualified, like `javax.servlet.Servlet`, when the package is not obvious from the context nor is it in one of the well known java packages like `java.lang`, `java.io`, `java.util` and `java.net`. Otherwise, the package is omitted like in `String`.

Exception and permission classes are not followed by the word "object". Readability is improved when the "object" suffix is avoided. For example, "to throw a `SecurityException`" and to "to have `FilePermission`" instead of "to have a `FilePermission` object".

Permissions can further be qualified with their actions. `ServicePermission[GET|REGISTER,com.acme.*]` means a `ServicePermission` with the action `GET` and `REGISTER` for all service names starting with `com.acme`. A `ServicePermission[REGISTER, Producer|Consumer]` means the `GET` `ServicePermission` for the `Producer` or `Consumer` class.

When discussing functionality of a class rather than the implementation details, the class name is written as normal text. This convention is often used when discussing services. For example, "the User Admin service".

Some services have the word "Service" embedded in their class name. In those cases, the word "service" is only used once but is written with an upper case S. For example, "the Log Service performs".

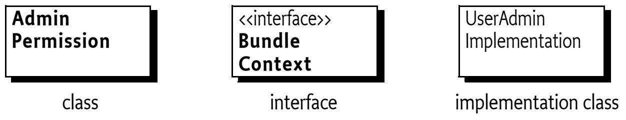
Service objects are registered with the OSGi Framework. Registration consists of the service object, a set of properties, and a list of classes and interfaces implemented by this service object. The classes and interfaces are used for type safety *and* naming. Therefore, it is said that a service object is registered *under* a class/interface. For example, "This service object is registered under `PermissionAdmin`."

1.4.3 Diagrams

The diagrams in this document illustrate the specification and are not normative. Their purpose is to provide a high-level overview on a single page. The following paragraphs describe the symbols and conventions used in these diagrams.

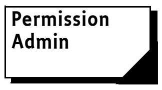
Classes or interfaces are depicted as rectangles, as in Figure 1. Interfaces are indicated with the qualifier <<interface>> as the first line. The name of the class/interface is indicated in bold when it is part of the specification. Implementation classes are sometimes shown to demonstrate a possible implementation. Implementation class names are shown in plain text. In certain cases class names are abbreviated. This is indicated by ending the abbreviation with a period.

Figure 1 *Class and interface symbol*



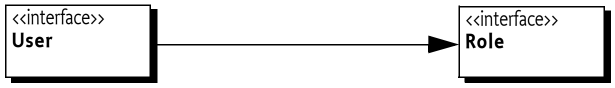
If an interface or class is used as a service object, it will have a black triangle in the bottom right corner.

Figure 2 *Service symbol*



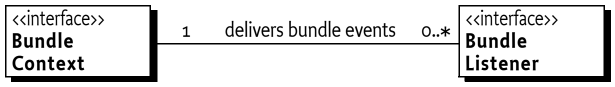
Inheritance (the extends or implements keyword in Java class definitions) is indicated with an arrow. Figure 3 shows that User implements or extends Role.

Figure 3 *Inheritance (implements or extends) symbol*



Relations are depicted with a line. The cardinality of the relation is given explicitly when relevant. Figure 4 shows that each (1) BundleContext object is related to 0 or more BundleListener objects, and that each BundleListener object is related to a single BundleContext object. Relations usually have some description associated with them. This description should be read from left to right and top to bottom, and includes the classes on both sides. For example: "A BundleContext object delivers bundle events to zero or more BundleListener objects."

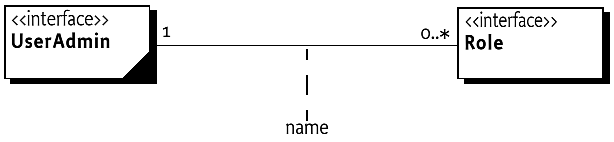
Figure 4 *Relations symbol*



Associations are depicted with a dashed line. Associations are between classes, and an association can be placed on a relation. For example, "every ServiceRegistration object has an associated ServiceReference object." This association does not have to be a hard relationship, but could be derived in some way.

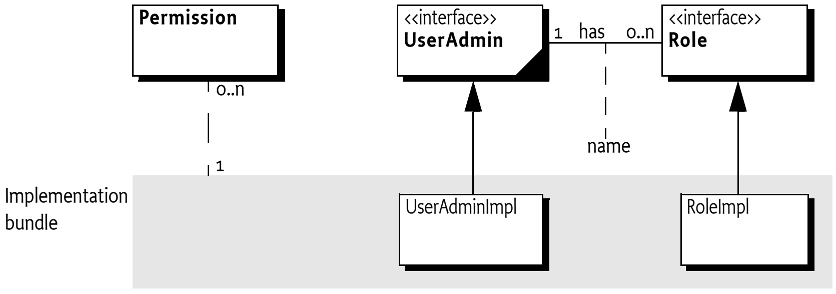
When a relationship is qualified by a name or an object, it is indicated by drawing a dotted line perpendicular to the relation and connecting this line to a class box or a description. Figure 5 shows that the relationship between a UserAdmin class and a Role class is qualified by a name. Such an association is usually implemented with a Dictionary object.

Figure 5 *Associations symbol*



Bundles are entities that are visible in normal application programming. For example, when a bundle is stopped, all its services will be unregistered. Therefore, the classes/interfaces that are grouped in bundles are shown on a grey rectangle.

Figure 6 *Bundles*



1.4.4 Key Words

This specification consistently uses the words *may*, *should*, and *must*. Their meaning is well defined in [1] *Bradner, S., Key words for use in RFCs to Indicate Requirement Levels*. A summary follows.

- *must* – An absolute requirement. Both the Framework implementation and bundles have obligations that are required to be fulfilled to conform to this specification.
- *should* – Recommended. It is strongly recommended to follow the description, but reasons may exist to deviate from this recommendation.
- *may* – Optional. Implementations must still be interoperable when these items are not implemented.

1.5 The Specification Process

Within the OSGi, specifications are developed by Expert Groups (EG). If a member company wants to participate in an EG, it must sign a Statement Of Work (SOW). The purpose of an SOW is to clarify the legal status of the material discussed in the EG. An EG will discuss material which already has

Intellectual Property (IP) rights associated with it, and may also generate new IP rights. The SOW, in conjunction with the member agreement, clearly defines the rights and obligations related to IP rights of the participants and other OSGi members.

To initiate work on a specification, a member company first submits a request for a proposal. This request is reviewed by the Market Requirement Committee which can either submit it to the Technical Steering Committee (TSC) or reject it. The TSC subsequently assigns the request to an EG to be implemented.

The EG will draft a number of proposals that meet the requirements from the request. Proposals usually contain Java code defining the API and semantics of the services under consideration. When the EG is satisfied with a proposal, it votes on it.

To assure that specifications can be implemented, reference implementations are created to implement the proposal. Test suites are also developed, usually by a different member company, to verify that the reference implementation (and future implementations by OSGi member companies) fulfill the requirements of the specifications. Reference implementations and test suites are *only* available to member companies.

Specifications combine a number of proposals to form a single document. The proposals are edited to form a set of consistent specifications, which are voted upon again by the EG. The specification is then submitted to all the member companies for review. During this review period, member companies must disclose any IP claims they have on the specification. After this period, the OSGi board of directors publishes the specification.

This Service Platform Release 3 specification was developed by the Core Platform Expert Group (CPEG), Device Expert Group (DEG), Remote Management Expert Group (RMEG), and Vehicle Expert Group (VEG).

1.6 Version Information

This document specifies OSGi Service Platform Release 3. This specification is backward compatible to releases 1 and 2.

New for this specification are:

- Wire Admin service
- Measurement utility
- Start Level service
- Execution Environments
- URL Stream and Content Handling
- Dynamic Import
- Position utility
- IO service
- XML service
- Jini service
- UPnP service
- OSGi Name-space
- Initial Provisioning service

Components in this specification have their own specification-version, independent of the OSGi Service Platform, Release 3 specification. The following table summarizes the packages and specification-versions for the different subjects.

Item	Package	Version
Framework	org.osgi.framework	1.2
Configuration Admin service	org.osgi.service.cm	1.1
Device Access	org.osgi.service.device	1.1
Http Service	org.osgi.service.http	1.1
IO Connector	org.osgi.service.io	1.0
Jini service	org.osgi.service.jini	1.0
Log Service	org.osgi.service.log	1.2
Metatype	org.osgi.service.metatype	1.0
Package Admin service	org.osgi.service.packageadmin	1.1
Permission Admin service	org.osgi.service.permissionadmin	1.1
Preferences Service	org.osgi.service.prefs	1.0
Initial Provisioning	org.osgi.service.provisioning	1.0
Bundle Start Levels	org.osgi.service.startlevel	1.0
Universal Plug & Play service	org.osgi.service.upnp	1.0
URL Stream and Content	org.osgi.service.url	1.0
User Admin service	org.osgi.service.useradmin	1.0
Wire Admin	org.osgi.service.wireadmin	1.0
Measurement utility	org.osgi.util.measurement	1.0
Position utility	org.osgi.util.position	1.0
Service Tracker	org.osgi.util.tracker	1.2
XML Parsers	org.osgi.util.xml	1.0

Table 1 Packages and versions

When a component is represented in a bundle, a specification-version is needed in the declaration of the Import-Package or Export-Package manifest headers. Package versioning is described in *Sharing Packages* on page 18.

1.7 Compliance Program

The OSGi offers a compliance program for the software product that includes an OSGi Framework and a set of zero or more core bundles collectively referred to as a Service Platform. Any services which exist in the org.osgi name-space and that are offered as part of a Service Platform must pass the conformance test suite in order for the product to be considered for inclusion in the compliance program. A Service Platform may be tested in isolation and is independent of its host Virtual Machine. Certification means that a product has passed the conformance test suite(s) and meets certain criteria necessary for admission to the program, including the requirement for the supplier to warrant and represent that the product conforms to the applicable OSGi specifications, as defined in the compliance requirements.

The compliance program is a voluntary program and participation is the supplier's option. The onus is on the supplier to ensure ongoing compliance with the certification program and any changes which may cause this compliance to be brought into question should result in re-testing and re-submission of the Service Platform. Only members of the OSGi alliance are permitted to submit certification requests.

1.7.1 Compliance Claims.

In addition, any product that contains a certified OSGi Service Platform may be said to contain an *OSGi Compliant Service Platform*. The product itself is not compliant and should not be claimed as such.

More information about the OSGi Compliance program, including the process for inclusion and the list of currently certified products, can be found at http://www.osgi.org/compliance.

1.8 References

[1] *Bradner, S., Key words for use in RFCs to Indicate Requirement Levels* http://www.ietf.org/rfc/rfc2119.txt, March 1997.

[2] *OSGi Service Gateway Specification 1.0* http://www.osgi.org/resources/spec_download.asp

[3] *OSGi Service Platform, Release 2, October 2001* http://www.osgi.org/resources/spec_download.asp

2 Log Service Specification

Version 1.2

2.1 Introduction

The Log Service provides a general purpose message logger for the OSGi Service Platform. It consists of two services, one for logging information and another for retrieving current or previously recorded log information.

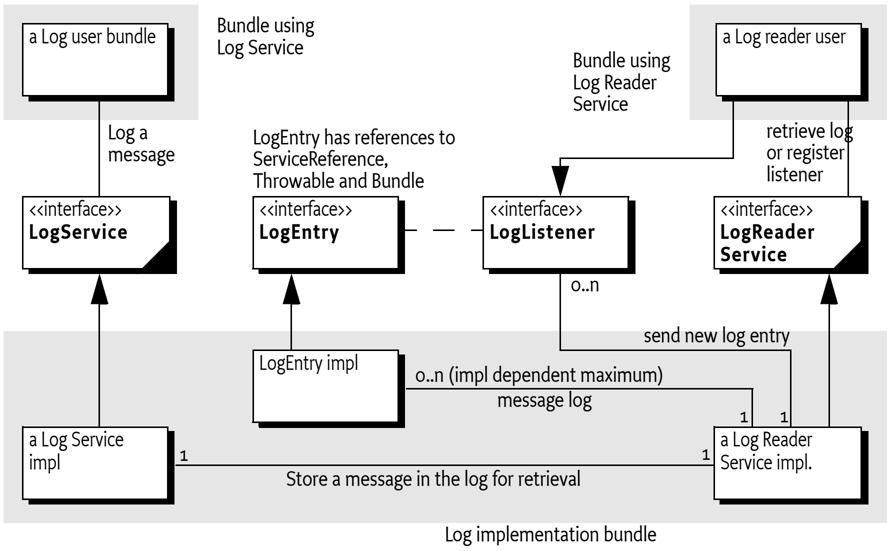
This specification defines the methods and semantics of interfaces which bundle developers can use to log entries and to retrieve log entries.

Bundles can use the Log Service to log information for the Operator. Other bundles, oriented toward management of the environment, can use the Log Reader Service to retrieve Log Entry objects that were recorded recently or to receive Log Entry objects as they are logged by other bundles.

2.1.1 Entities

- *LogService* – The service interface that allows a bundle to log information, including a message, a level, an exception, a ServiceReference object, and a Bundle object.
- *LogEntry* - An interface that allows access to a log entry in the log. It includes all the information that can be logged through the Log Service and a time stamp.
- *LogReaderService* - A service interface that allows access to a list of recent LogEntry objects, and allows the registration of a LogListener object that receives LogEntry objects as they are created.
- *LogListener* - The interface for the listener to LogEntry objects. Must be registered with the Log Reader Service.

Figure 7 Log Service Class Diagram org.osgi.service.log package


a Log user bundle
Bundle using Log Service
Bundle using Log Reader Service
a Log reader user
Log a message
LogEntry has references to ServiceReference, Throwable and Bundle
retrieve log or register listener
<<interface>> LogService
<<interface>> LogEntry
<<interface>> LogListener
<<interface>> LogReader Service
0..n
send new log entry
LogEntry impl
0..n (impl dependent maximum) message log
1
1
a Log Service impl
1
1
a Log Reader Service impl.
Store a message in the log for retrieval
Log implementation bundle

2.2 The Log Service Interface

The LogService interface allows bundle developers to log messages that can be distributed to other bundles, which in turn can forward the logged entries to a file system, remote system, or some other destination.

The LogService interface allows the bundle developer to:

- Specify a message and/or exception to be logged.
- Supply a log level representing the severity of the message being logged. This should be one of the levels defined in the LogService interface but it may be any integer that is interpreted in a user-defined way.
- Specify the Service associated with the log requests.

By obtaining a LogService object from the Framework service registry, a bundle can start logging messages to the LogService object by calling one of the LogService methods. A Log Service object can log any message, but it is primarily intended for reporting events and error conditions.

The LogService interface defines these methods for logging messages:

- log(int, String) – This method logs a simple message at a given log level.
- log(int, String, Throwable) – This method logs a message with an exception at a given log level.
- log(ServiceReference, int, String) – This method logs a message associated with a specific service.
- log(ServiceReference, int, String, Throwable) – This method logs a message with an exception associated with a specific service.

While it is possible for a bundle to call one of the log methods without providing a ServiceReference object, it is recommended that the caller supply the ServiceReference argument whenever appropriate, because it provides important context information to the operator in the event of problems.

The following example demonstrates the use of a log method to write a message into the log.

```
logService.log(
   myServiceReference,
   LogService.LOG_INFO,
   "myService is up and running"
);
```

In the example, the myServiceReference parameter identifies the service associated with the log request. The specified level, LogService.LOG_INFO, indicates that this message is informational.

The following example code records error conditions as log messages.

```
try {
   FileInputStream fis = new FileInputStream("myFile");
   int b;
   while ( (b = fis.read()) != -1 ) {
      ...
   }
   fis.close();
}
catch ( IOException exception ) {
   logService.log(
      myServiceReference,
      LogService.LOG_ERROR,
      "Cannot access file",
      exception );
}
```

Notice that in addition to the error message, the exception itself is also logged. Providing this information can significantly simplify problem determination by the Operator.

2.3 Log Level and Error Severity

The log methods expect a log level indicating error severity, which can be used to filter log messages when they are retrieved. The severity levels are defined in the LogService interface.

Callers must supply the log levels that they deem appropriate when making log requests. The following table lists the log levels.

Level	Descriptions
LOG_DEBUG	Used for problem determination and may be irrelevant to anyone but the bundle developer.
LOG_ERROR	Indicates the bundle or service may not be functional. Action should be taken to correct this situation.

Table 2 *Log Levels*

Level	Descriptions
LOG_INFO	May be the result of any change in the bundle or service and does not indicate a problem.
LOG_WARNING	Indicates a bundle or service is still functioning but may experience problems in the future because of the warning condition.

Table 2 Log Levels

2.4 Log Reader Service

The Log Reader Service maintains a list of LogEntry objects called the *log*. The Log Reader Service is a service that bundle developers can use to retrieve information contained in this log, and receive notifications about LogEntry objects when they are created through the Log Service.

The size of the log is implementation-specific, and it determines how far into the past the log entries go. Additionally, some log entries may not be recorded in the log in order to save space. In particular, LOG_DEBUG log entries may not be recorded. Note that this rule is implementation-dependent. Some implementations may allow a configurable policy to ignore certain LogEntry object types.

The LogReaderService interface defines these methods for retrieving log entries.

- getLog() – This method retrieves past log entries as an enumeration with the most recent entry first.
- addLogListener(LogListener) – This method is used to subscribe to the Log Reader Service in order to receive log messages as they occur. Unlike the previously recorded log entries, all log messages must be sent to subscribers of the Log Reader Service as they are recorded.
  A subscriber to the Log Reader Service must implement the LogListener interface.
  After a subscription to the Log Reader Service has been started, the subscriber's LogListener.logged method must be called with a LogEntry object for the message each time a message is logged.

The LogListener interface defines the following method:

- logged(LogEntry) – This method is called for each LogEntry object created. A Log Reader Service implementation must not filter entries to the LogListener interface as it is allowed to do for its log. A LogListener object should see all LogEntry objects that are created.

The delivery of LogEntry objects to the LogListener object should be done asynchronously.

2.5 Log Entry Interface

The LogEntry interface abstracts a log entry. It is a record of the information that was passed when an event was logged, and consists of a superset of information which can be passed through the LogService methods. The LogEntry interface defines these methods to retrieve information related to LogEntry objects:

- getBundle() – This method returns the Bundle object related to a Log-Entry object.
- getException() – This method returns the exception related to a Log-Entry object. In some implementations, the returned exception may not be the original exception. To avoid references to a bundle defined exception class, thus preventing an uninstalled bundle from being garbage collected, the Log Service may return an exception object of an implementation defined Throwable subclass. This object will attempt to return as much information as possible, such as the message and stack trace, from the original exception object .
- getLevel() – This method returns the severity level related to a LogEntry object.
- getMessage() – This method returns the message related to a LogEntry object.
- getServiceReference() –This method returns the ServiceReference object of the service related to a LogEntry object.
- getTime() – This method returns the time that the log entry was created.

2.6 Mapping of Events

Implementations of a Log Service must log Framework-generated events and map the information to LogEntry objects in a consistent way. Framework events must be treated exactly the same as other logged events and distributed to all LogListener objects that are associated with the Log Reader Service. The following sections define the mapping for the three different event types: Bundle, Service, and Framework.

2.6.1 Bundle Events Mapping

A Bundle Event is mapped to a LogEntry object according to Table 3, “Mapping of Bundle Events to Log Entries,” on page 15.

Log Entry method	Information about Bundle Event
getLevel()	LOG_INFO
getBundle()	Identifies the bundle to which the event happened. In other words, it identifies the bundle that was installed, started, stopped, updated, or uninstalled. This identification is obtained by calling getBundle() on the BundleEvent object.
getException()	null

Table 3 Mapping of Bundle Events to Log Entries

Log Entry method	Information about Bundle Event
getServiceReference()	null
getMessage()	The message depends on the event type: • INSTALLED – "BundleEvent INSTALLED" • STARTED – "BundleEvent STARTED" • STOPPED – "BundleEvent STOPPED" • UPDATED – "BundleEvent UPDATED" • UNINSTALLED – "BundleEvent UNINSTALLED"

Table 3 Mapping of Bundle Events to Log Entries

2.6.2 Service Events Mapping

A Service Event is mapped to a LogEntry object according to Table 4, "Mapping of Service Events to Log Entries," on page 16.

Log Entry method	Information about Service Event
getLevel()	LOG_INFO, except for the ServiceEvent.MODIFIED event. This event can happen frequently and contains relatively little information. It must be logged with a level of LOG_DEBUG.
getBundle()	Identifies the bundle that registered the service associated with this event. It is obtained by calling getServiceReference().getBundle() on the ServiceEvent object.
getException()	null
getServiceReference()	Identifies a reference to the service associated with the event. It is obtained by calling getServiceReference() on the ServiceEvent object.
getMessage()	This message depends on the actual event type. The messages are mapped as follows: • REGISTERED – "ServiceEvent REGISTERED" • MODIFIED – "ServiceEvent MODIFIED" • UNREGISTERING – "ServiceEvent UNREGISTERING"

Table 4 Mapping of Service Events to Log Entries

2.6.3 Framework Events Mapping

A Framework Event is mapped to a LogEntry object according to Table 5, "Mapping of Framework Event to Log Entries," on page 17.

Log Entry method	Information about Framework Event
getLevel()	LOG_INFO, except for the FrameworkEvent.ERROR event. This event represents an error and is logged with a level of LOG_ERROR.
getBundle()	Identifies the bundle associated with the event. This may be the system bundle. It is obtained by calling getBundle() on the FrameworkEvent object.
getException()	Identifies the exception associated with the error. This will be null for event types other than ERROR. It is obtained by calling getThrowable() on the FrameworkEvent object.
getServiceReference()	null
getMessage()	This message depends on the actual event type. The messages are mapped as follows: • STARTED – "FrameworkEvent STARTED" • ERROR – "FrameworkEvent ERROR" • PACKAGES_REFRESHED – "FrameworkEvent PACKAGES REFRESHED" • STARTLEVEL_CHANGED – "FrameworkEvent STARTLEVEL CHANGED"

Table 5 *Mapping of Framework Event to Log Entries*

2.7 Security

The Log Service should only be implemented by trusted bundles. This bundle requires ServicePermission[REGISTER,LogService|LogReaderService]. Virtually all bundles should get ServicePermission[GET,LogService]. The ServicePermission[GET,LogReaderService] should only be assigned to trusted bundles.

2.8 Changes

The following clarifications were made.

- The interpretation of the log level has been clarified to allow arbitrary integers.
- New Framework Event type strings are defined.
- LogEntry.getException is allowed to return a different exception object than the original exception object in order to allow garbage collection of the original object.
- The addLogListener method in the Log Reader Service no longer adds the same listener object twice.
- Delivery of Log Event objects to Log Listener objects must happen asynchronously. This delivery mode was undefined in previous releases.

2.9 org.osgi.service.log

The OSGi Log Service Package. Specification Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.log; specification-ver-
sion=1.2
```

2.9.1 Summary

- LogEntry - Provides methods to access the information contained in an individual Log Service log entry. [p.11]
- LogListener - Subscribes to `LogEntry` objects from the `LogReaderService`. [p.11]
- LogReaderService - Provides methods to retrieve `LogEntry` objects from the log. [p.11]
- LogService - Provides methods for bundles to write messages to the log. [p.11]

2.9.2 public interface LogEntry

Provides methods to access the information contained in an individual Log Service log entry.

A `LogEntry` object may be acquired from the `LogReaderService.getLog` method or by registering a `LogListener` object.

See Also `LogReaderService.getLog`[p.20] , `LogListener`[p.11]

2.9.2.1 public Bundle getBundle( )

□ Returns the bundle that created this `LogEntry` object.

Returns The bundle that created this `LogEntry` object; `null` if no bundle is associated with this `LogEntry` object.

2.9.2.2 public Throwable getException( )

□ Returns the exception object associated with this `LogEntry` object.

In some implementations, the returned exception may not be the original exception. To avoid references to a bundle defined exception class, thus preventing an uninstalled bundle from being garbage collected, the Log Service may return an exception object of an implementation defined Throwable subclass. The returned object will attempt to provide as much information as possible from the original exception object such as the message and stack trace.

Returns `Throwable` object of the exception associated with this `LogEntry`;`null` if no exception is associated with this `LogEntry` object.

2.9.2.3 public int getLevel( )

□ Returns the severity level of this `LogEntry` object.

This is one of the severity levels defined by the `LogService` interface.

Returns Severity level of this `LogEntry` object.

See Also LogService.LOG_ERROR[p.21], LogService.LOG_WARNING[p.21], LogService.LOG_INFO[p.21], LogService.LOG_DEBUG[p.20]

2.9.2.4 public String getMessage( )

□ Returns the human readable message associated with this LogEntry object.

Returns String containing the message associated with this LogEntry object.

2.9.2.5 public ServiceReference getServiceReference( )

□ Returns the ServiceReference object for the service associated with this LogEntry object.

Returns ServiceReference object for the service associated with this LogEntry object; null if no ServiceReference object was provided.

2.9.2.6 public long getTime( )

□ Returns the value of currentTimeMillis() at the time this LogEntry object was created.

Returns The system time in milliseconds when this LogEntry object was created.

See Also System.currentTimeMillis()

2.9.3 public interface LogListener extends EventListener

Subscribes to LogEntry objects from the LogReaderService.

A LogListener object may be registered with the Log Reader Service using the LogReaderService.addLogListener method. After the listener is registered, the logged method will be called for each LogEntry object created. The LogListener object may be unregistered by calling the LogReaderService.removeLogListener method.

See Also LogReaderService[p.11], LogEntry[p.11], LogReaderService.addLogListener(LogListener)[p.20], LogReaderService.removeLogListener(LogListener)[p.20]

2.9.3.1 public void logged(LogEntry entry)

entry A LogEntry object containing log information.

□ Listener method called for each LogEntry object created.

As with all event listeners, this method should return to its caller as soon as possible.

See Also LogEntry[p.11]

2.9.4 public interface LogReaderService

Provides methods to retrieve LogEntry objects from the log.

There are two ways to retrieve LogEntry objects:

- The primary way to retrieve LogEntry objects is to register a LogListener object whose LogListener.logged method will be called for each entry added to the log.
- To retrieve past LogEntry objects, the getLog method can be called which will return an Enumeration of all LogEntry objects in the log.

See Also LogEntry[p.11], LogListener[p.11], LogListener.logged(LogEntry) [p.19]

2.9.4.1 public void addLogListener(LogListener listener)

listener A LogListener object to register; the LogListener object is used to receive LogEntry objects.

□ Subscribes to LogEntry objects.

This method registers a LogListener object with the Log Reader Service. The LogListener.logged(LogEntry) method will be called for each LogEntry object placed into the log.

When a bundle which registers a LogListener object is stopped or otherwise releases the Log Reader Service, the Log Reader Service must remove all of the bundle's listeners.

If this Log Reader Service's list of listeners already contains a listener l such that (l==listener), this method does nothing.

See Also LogListener[p.11], LogEntry[p.11], LogListener.logged(LogEntry) [p.19]

2.9.4.2 public Enumeration getLog( )

□ Returns an Enumeration of all LogEntry objects in the log.

Each element of the enumeration is a LogEntry object, ordered with the most recent entry first. Whether the enumeration is of all LogEntry objects since the Log Service was started or some recent past is implementation-specific. Also implementation-specific is whether informational and debug LogEntry objects are included in the enumeration.

2.9.4.3 public void removeLogListener(LogListener listener)

listener A LogListener object to unregister.

□ Unsubscribes to LogEntry objects.

This method unregisters a LogListener object from the Log Reader Service.

If listener is not contained in this Log Reader Service's list of listeners, this method does nothing.

See Also LogListener[p.11]

2.9.5 public interface LogService

Provides methods for bundles to write messages to the log.

LogService methods are provided to log messages; optionally with a ServiceReference object or an exception.

Bundles must log messages in the OSGi environment with a severity level according to the following hierarchy:

1 LOG_ERROR[p.21]
2 LOG_WARNING[p.21]
3 LOG_INFO[p.21]
4 LOG_DEBUG[p.20]

2.9.5.1 public static final int LOG_DEBUG = 4

A debugging message (Value 4).

This log entry is used for problem determination and may be irrelevant to anyone but the bundle developer.

2.9.5.2 public static final int LOG_ERROR = 1

An error message (Value 1).

This log entry indicates the bundle or service may not be functional.

2.9.5.3 public static final int LOG_INFO = 3

An informational message (Value 3).

This log entry may be the result of any change in the bundle or service and does not indicate a problem.

2.9.5.4 public static final int LOG_WARNING = 2

A warning message (Value 2).

This log entry indicates a bundle or service is still functioning but may experience problems in the future because of the warning condition.

2.9.5.5 public void log(int level, String message)

level The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.

message Human readable string describing the condition or `null`.

□ Logs a message.

The `ServiceReference` field and the `Throwable` field of the `LogEntry` object will be set to `null`.

See Also `LOG_ERROR`[p.21] , `LOG_WARNING`[p.21] , `LOG_INFO`[p.21] , `LOG_DEBUG`[p.20]

2.9.5.6 public void log(int level, String message, Throwable exception)

level The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.

message The human readable string describing the condition or `null`.

exception The exception that reflects the condition or `null`.

□ Logs a message with an exception.

The `ServiceReference` field of the `LogEntry` object will be set to `null`.

See Also `LOG_ERROR`[p.21] , `LOG_WARNING`[p.21] , `LOG_INFO`[p.21] , `LOG_DEBUG`[p.20]

2.9.5.7 public void log(ServiceReference sr, int level, String message)

sr The `ServiceReference` object of the service that this message is associated with or `null`.

level The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.

message Human readable string describing the condition or `null`.

□ Logs a message associated with a specific `ServiceReference` object.

The `Throwable` field of the `LogEntry` will be set to `null`.

See Also `LOG_ERROR`[p.21] , `LOG_WARNING`[p.21] , `LOG_INFO`[p.21] , `LOG_DEBUG`[p.20]

2.9.5.8 public void log(ServiceReference sr, int level, String message, Throwable exception)

sr	The `ServiceReference` object of the service that this message is associated with.
level	The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
message	Human readable string describing the condition or `null`.
exception	The exception that reflects the condition or `null`.

□ Logs a message with an exception associated and a `ServiceReference` object.

See Also `LOG_ERROR`[p.21] , `LOG_WARNING`[p.21] , `LOG_INFO`[p.21] , `LOG_DEBUG`[p.20]

3 Configuration Admin Service Specification

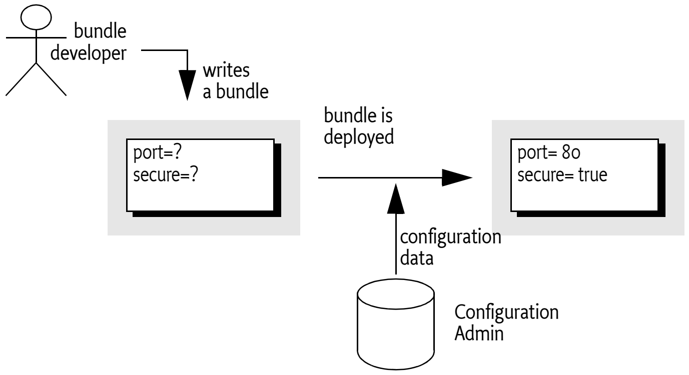
Version 1.1

3.1 Introduction

The Configuration Admin service is an important aspect of the deployment of an OSGi Service Platform. It allows an Operator to set the configuration information of deployed bundles.

Configuration is the process of defining the configuration data of bundles and assuring that those bundles receive that data when they are active in the OSGi Service Platform.

Figure 8 *Configuration Admin Service Overview*



3.1.1 Essentials

The following requirements and patterns are associated with the Configuration Admin service specification:

- *Local Configuration* – The Configuration Admin service must support bundles that have their own user interface to change their configurations.
- *Reflection* – The Configuration Admin service must be able to deduce the names and types of the needed configuration data.
- *Legacy* – The Configuration Admin service must support configuration data of existing entities (such as devices).
- *Object Oriented* – The Configuration Admin service must support the creation and deletion of instances of configuration information so that a bundle can create the appropriate number of services under the control of the Configuration Admin service.

- *Embedded Devices* – The Configuration Admin service must be deployable on a wide range of platforms. This requirement means that the interface should not assume file storage on the platform. The choice to use file storage should be left to the implementation of the Configuration Admin service.
- *Remote versus Local Management* – The Configuration Admin service must allow for a remotely managed OSGi Service Platform, and must not assume that configuration information is stored locally. Nor should it assume that the Configuration Admin service is always done remotely. Both implementation approaches should be viable.
- *Availability* – The OSGi environment is a dynamic environment that must run continuously (24/7/365). Configuration updates must happen dynamically and should not require restarting of the system or bundles.
- *Immediate Response* – Changes in configuration should be reflected immediately.
- *Execution Environment* – The Configuration Admin service will not require more than an environment that fulfills the minimal execution requirements.
- *Communications* – The Configuration Admin service should not assume "always-on" connectivity, so the API is also applicable for mobile applications in cars, phones, or boats.
- *Extendability* – The Configuration Admin service should expose the process of configuration to other bundles. This exposure should at a minimum encompass initiating an update, removing certain configuration properties, adding properties, and modifying the value of properties potentially based on existing property or service values.
- *Complexity Trade-offs* – Bundles in need of configuration data should have a simple way of obtaining it. Most bundles have this need and the code to accept this data. Additionally, updates should be simple from the perspective of the receiver.
  Trade-offs in simplicity should be made at the expense of the bundle implementing the Configuration Admin service and in favor of bundles that need configuration information. The reason for this choice is that normal bundles will outnumber Configuration Admin bundles.

3.1.2 Operation

This specification is based on the concept of a Configuration Admin service that manages the configuration of an OSGi Service Platform. It maintains a database of `Configuration` objects, locally or remote. This service monitors the service registry and provides configuration information to services that are registered with a `service.pid` property, the Persistent IDentity (PID), and implement one of the following interfaces:

- *Managed Service* – A service registered with this interface receives its *configuration dictionary* from the database or receives null when no such configuration exists or when an existing configuration has never been updated.
- *Managed Service Factory* – Services registered with this interface receive several configuration dictionaries when registered. The database contains zero or more configuration dictionaries for this service. Each configuration dictionary is given sequentially to the service.

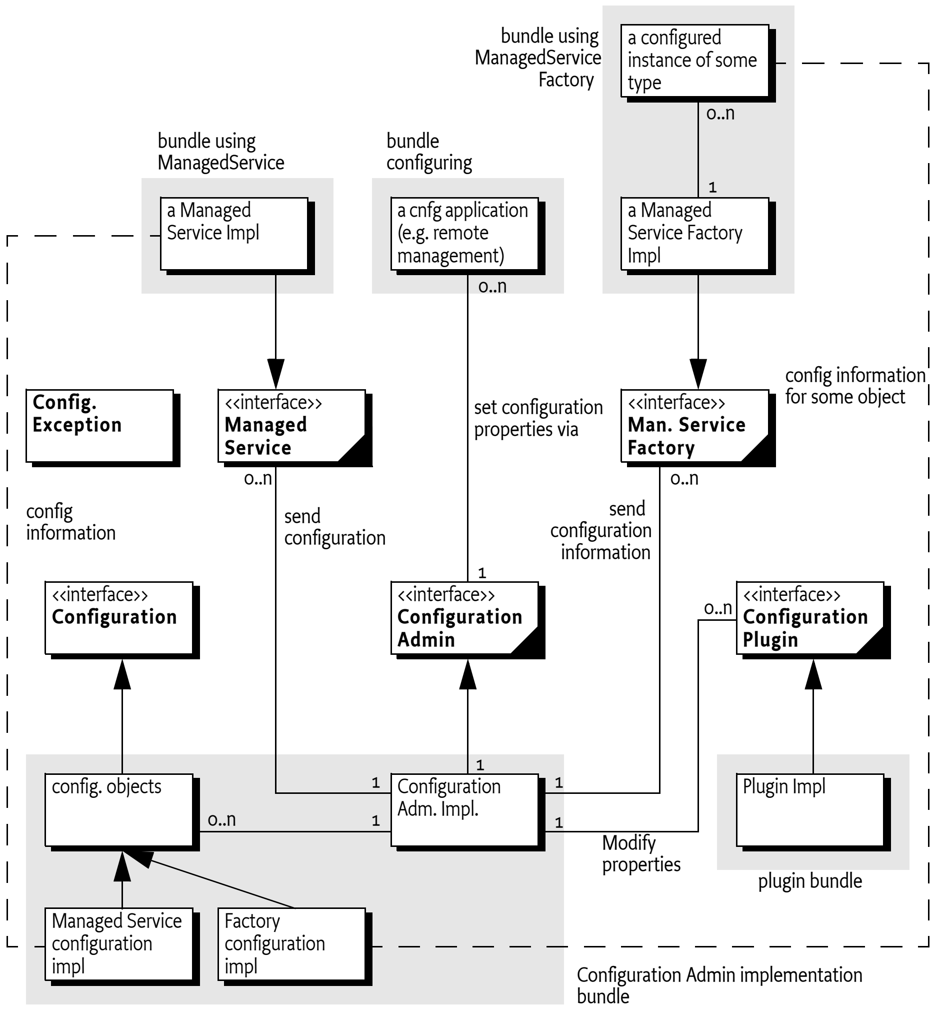
The database can be manipulated either by the Management Agent or bundles that configure themselves.

Other parties can provide Configuration Plugin services. Such services participate in the configuration process. They can inspect the configuration dictionary and modify it before it reaches the target service.

3.1.3 Entities

- *Configuration information* – The information needed by a bundle before it can provide its intended functionality.
- *Configuration dictionary* – The configuration information when it is passed to the target service. It consists of a Dictionary object with a number of properties and identifiers.
- *Configuring Bundle* – A bundle that modifies the configuration information through the Configuration Admin service. This bundle is either a management bundle or the bundle for which the configuration information is intended.
- *Configuration Target* – The target (bundle or service) that will receive the configuration information. For services, there are two types of targets: ManagedServiceFactory or ManagedService objects.
- *Configuration Admin Service* – This service is responsible for supplying configuration target bundles with their configuration information. It maintains a database with configuration information, keyed on the service.pid of configuration target services. These services receive their configuration dictionary or dictionaries when they are registered with the Framework. Configurations can be modified or extended using Configuration Plugin services before they reach the target bundle.
- *Managed Service* – A Managed Service represents a client of the Configuration Admin service, and is thus a configuration target. Bundles should register a Managed Service to receive the configuration data from the Configuration Admin service. A Managed Service adds a unique service.pid service registration property as a primary key for the configuration information.
- *Managed Service Factory* – A Managed Service Factory can receive a number of configuration dictionaries from the Configuration Admin service, and is thus also a configuration target service. It should register with a service.pid and receives zero or more configuration dictionaries. Each dictionary has its own PID.
- *Configuration Object* – Implements the Configuration interface and contains the configuration dictionary for a Managed Service or one of the configuration dictionaries for a Managed Service Factory. These objects are manipulated by configuring bundles.
- *Configuration Plugin* Services – Configuration Plugin services are called before the configuration dictionary is given to the configuration targets. The plug-in can modify the configuration dictionary, which is passed to the Configuration Target.

Figure 9 Configuration Admin Class Diagram org.osgi.service.cm


bundle using ManagedService Factory
a configured instance of some type
0..n
1
bundle using ManagedService
bundle configuring
a Managed Service Impl
a cnfg application (e.g. remote management)
0..n
a Managed Service Factory Impl
config information for some object
Config. Exception
<<interface>> Managed Service
set configuration properties via
<<interface>> Man. Service Factory
0..n
0..n
config information
send configuration
send configuration information
1
<<interface>> Configuration
<<interface>> Configuration Admin
0..n
<<interface>> Configuration Plugin
config. objects
0..n
1
1
Configuration Adm. Impl.
1
1
1
Plugin Impl
Modify properties
plugin bundle
Managed Service configuration impl
Factory configuration impl
Configuration Admin implementation bundle

3.2 Configuration Targets

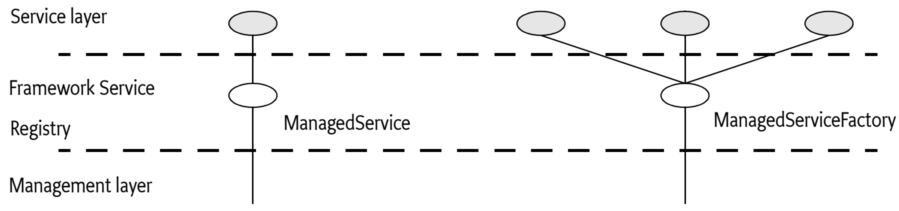
One of the more complicated aspects of this specification is the subtle distinction between the ManagedService and ManagedServiceFactory classes.

Both receive configuration information from the Configuration Admin service and are treated similarly in most respects. Therefore, this specification refers to *configuration targets* when the distinction is irrelevant.

The difference between these types is related to the cardinality of the configuration dictionary. A Managed Service is used when an existing entity needs a configuration dictionary. Thus, a one-to-one relationship always exists between the configuration dictionary and the entity.

A Managed Service Factory is used when part of the configuration is to define *how many instances are required*. A management bundle can create, modify, and delete any number of instances for a Managed Service Factory through the Configuration Admin service. Each instance is configured by a single `Configuration` object. Therefore, a Managed Service Factory can have multiple associated `Configuration` objects.

Figure 10 *Differentiation of ManagedService and ManagedServiceFactory Classes*



To summarize:

- A *Managed Service* must receive a single configuration dictionary when it is registered or when its configuration is modified.
- A *Managed Service Factory* must receive from zero to *n* configuration dictionaries when it registers, depending on the current configuration. The Managed Service Factory is informed of configuration dictionary changes: modifications, creations, and deletions.

3.3 The Persistent Identity

A crucial concept in the Configuration Admin service specification is the Persistent IDentity (PID). Its purpose is to act as a primary key for objects that need a configuration dictionary. The name of the service property for PID is defined in the Framework in `org.osgi.framework.Constants.SERVICE_PID`.

A PID is a unique identifier for a service that persists over multiple invocations of the Framework.

When a bundle registers a service with a PID, it should set property `service.pid` to a unique value. For that service, the same PID should always be used. If the bundle is stopped and later started, the same PID should be used.

PIDs can be useful for all services, but the Configuration Admin service requires their use with Managed Service and Managed Service Factory registrations because it associates its configuration data with PIDs.

PIDs must be unique for each service. A bundle must not register multiple configuration target services with the same PID. If that should occur, the Configuration Admin service must:

- Send the appropriate configuration data to all services registered under that PID from that bundle only.
- Report an error in the log.
- Ignore duplicate PIDs from other bundles and report them to the log.

3.3.1 PID Syntax

PIDs are intended for use by other bundles, not by people, but sometimes the user is confronted with a PID. For example, when installing an alarm system, the user needs to identify the different components to a wiring application. This type of application exposes the PID to end users.

The schemes for PIDs that are defined in this specification should be followed.

Any globally unique string can be used as a PID. The following sections, however, define schemes for common cases. These schemes are not required, but bundle developers are urged to use them to achieve consistency.

3.3.1.1 Local Bundle PIDs

As a convention, descriptions starting with the bundle identity and a dot (.) are reserved for a bundle. As an example, a PID of "65.536" would belong to the bundle with a bundle identity of 65.

3.3.1.2 Software PIDs

Configuration target services that are singletons can use a Java package name they own as the PID (the reverse domain name scheme). As an example, the PID named com.acme.watchdog would represent a Watchdog service from the ACME company.

3.3.1.3 Devices

Devices are usually organized on buses or networks. The identity of a device, such as a unique serial number or an address, is a good component of a PID. The format of the serial number should be the same as that printed on the housing or box, to aid in recognition..

Bus	Example	Format	Description
USB	USB-0123-0002-9909873	idVendor (hex 4) idProduct (hex 4) iSerialNumber (decimal)	Universal Serial Bus. Use the standard device descriptor.
IP	IP-172.16.28.21	IP nr (dotted decimal)	Internet Protocol
802	802-00:60:97:00:9A:56	MAC address with : separators	IEEE 802 MAC address (Token Ring, Ethernet,...)
ONE	ONE-06-00000021E461	Family (hex 2) and serial number including CRC (hex 6)	1-wire bus of Dallas Semiconductor
COM	COM-krups-brewer-12323	serial number or type name of device	Serial ports

Table 6 Schemes for Device-Oriented PID Names

3.4 The Configuration Object

A Configuration object contains the configuration dictionary, which is a set of properties that configure an aspect of a bundle. A bundle can receive Configuration objects by registering a configuration target service with a PID service property. See *The Persistent Identity* on page 27 for more information about PIDs.

During registration, the Configuration Admin service must detect these configuration target services and hand over their configuration dictionary via a callback. If this configuration dictionary is subsequently modified, the modified dictionary is handed over to the configuration target again with the same callback.

The Configuration object is primarily a set of properties that can be updated by a Management Agent, user interfaces on the OSGi Service Platform, or other applications. Configuration changes are first made persistent, and then passed to the target service via a call to the updated method in the ManagedServiceFactory or ManagedService class.

A Configuration object must be uniquely bound to a Managed Service or Managed Service Factory. This implies that a bundle must not register a Managed Service Factory with a PID that is the same as the PID given to a Managed Service.

3.4.1 Location Binding

When a Configuration object is created by either getConfiguration or createFactoryConfiguration, it becomes bound to the location of the calling bundle. This location is obtained with the associated bundle's getLocation method.

Location binding is a security feature that assures that only management bundles can modify configuration data, and other bundles can only modify their own configuration data. A SecurityException is thrown if a bundle other than a Management Agent bundle attempts to modify the configuration information of another bundle.

If a Managed Service is registered with a PID that is already bound to another location, the normal callback to ManagedService.updated must not take place.

The two argument versions of getConfiguration and createFactoryConfiguration take a location String as their second argument. These methods require AdminPermission, and they create Configuration objects bound to the specified location, instead of the location of the calling bundle. These methods are intended for management bundles.

The creation of a Configuration object does not in itself initiate a callback to the target.

A null location parameter may be used to create Configuration objects that are not bound. In this case, the objects become bound to a specific location the first time that they are used by a bundle. When this dynamically bound bundle is subsequently uninstalled, the Configuration object's bundle location must be set to null again so it can be bound again later.

A management bundle may create a Configuration object before the associated Managed Service is registered. It may use a null location to avoid any dependency on the actual location of the bundle which registers this service. When the Managed Service is registered later, the Configuration object must be bound to the location of the registering bundle, and its configuration dictionary must then be passed to ManagedService.updated.

3.4.2 Configuration Properties

A configuration dictionary contains a set of properties in a Dictionary object. The value of the property may be of the following types:

```
type        ::=
   String     | Integer    | Long
 | Float      | Double     | Byte
 | Short      | Character  | Boolean
 | vector     | arrays

primitive   ::= long | int | short | char | byte | double
 | float | boolean

arrays      ::= primitive '[]' | type '[]' | null

vector      ::= Vector of type or null
```

This explicitly allows vectors and arrays of mixed types and containing null.

The name or key of a property must always be a String object, and is not case sensitive during look up, but must preserve the original case.

Bundles should not use nested vectors or arrays, nor should they use mixed types. Using mixed types or nesting makes it impossible to use the meta typing specification. See *Metatype Specification* on page 65.

3.4.3 Property Propagation

An implementation of a Managed Service should copy all the properties of the Dictionary object argument in updated(Dictionary), known or unknown, into its service registration properties using ServiceRegistration.setProperties.

This propagation allows the development of applications that leverage the Framework service registry more extensively, so compliance with this mechanism is advised.

A configuration target service may ignore any configuration properties it does not recognize, or it may change the values of the configuration properties before these properties are registered. Configuration properties in the Framework service registry are not strictly related to the configuration information.

Bundles that cooperate with the propagation of configuration properties can participate in horizontal applications. For example, an application that maintains physical location information in the Framework service registry could find out where a particular device is located in the house or car. This service could use a property dedicated to the physical location and provide functions that leverage this property, such as a graphic user interface that displays these locations.

3.4.4 Automatic Properties

The Configuration Admin service must automatically add a number of properties to the configuration dictionary. If these properties are also set by a configuring bundle or a plug-in, they must always be overridden before they are given to the target service. See *Configuration Plugin* on page 42, Therefore, the receiving bundle or plug-in can assume that the following properties are defined by the Configuration Admin service and not by the configuring bundle:

- service.pid – Set to the PID of the associated Configuration object.
- service.factoryPid – Only set for a Managed Service Factory. It is then set to the PID of the associated Managed Service Factory.
- service.bundleLocation – Set to the location of the bundle that can use this Configuration object. This property can only be used for searching, it may not appear in the configuration dictionary returned from the getProperties method due to security reasons, nor may it be used when the target is updated.

Constants for some of these properties can be found in org.osgi.framework.Constants. These system properties are all of type String.

3.4.5 Equality

Two different Configuration objects can actually represent the same underlying configuration. This means that a Configuration object must implement the equals and hashCode methods in such a way that two Configuration objects are equal when their PID is equal.

3.5 Managed Service

A Managed Service is used by a bundle that needs one configuration dictionary and is thus associated with one Configuration object in the Configuration Admin service.

A bundle can register any number of ManagedService objects, but each must be identified with its own PID.

A bundle should use a Managed Service when it needs configuration information for the following:

- *A Singleton* – A single entity in the bundle that needs to be configured.
- *Externally Detected Devices* – Each device that is detected causes a registration of an associated ManagedService object. The PID of this object is related to the identity of the device, such as the address or serial number.

3.5.1 Networks

When a device in the external world needs to be represented in the OSGi Environment, it must be detected in some manner. The Configuration Admin service cannot know the identity and the number of instances of the device without assistance. When a device is detected, it still needs configuration information in order to play a useful role.

For example, a 1-Wire network can automatically detect devices that are attached and removed. When it detects a temperature sensor, it could register a Sensor service with the Framework service registry. This Sensor service needs configuration information specifically for that sensor, such as which lamps should be turned on, at what temperature the sensor is triggered, what timer should be started, in what zone it resides, and so on. One bundle could potentially have hundreds of these sensors and actuators, and each needs its own configuration information.

Each of these Sensor services should be registered as a Managed Service with a PID related to the physical sensor (such as the address) to receive configuration information.

Other examples are services discovered on networks with protocols like Jini, UPnP, and Salutation. They can usually be represented in the Framework service registry. A network printer, for example, could be detected via UPnP. Once in the service registry, these services usually require local configuration information. A Printer service needs to be configured for its local role: location, access list, and so on.

This information needs to be available in the Framework service registry whenever that particular Printer service is registered. Therefore, the Configuration Admin service must remember the configuration information for this Printer service.

This type of service should register with the Framework as a Managed Service in order to receive appropriate configuration information.

3.5.2 Singletons

When an object must be instantiated only once, it is called a *singleton*. A singleton requires a single configuration dictionary. Bundles may implement several different types of singletons if necessary.

For example, a Watchdog service could watch the registry for the status and presence of services in the Framework service registry. Only one instance of a Watchdog service is needed, so only a single configuration dictionary is required that contains the polling time and the list of services to watch.

3.5.3 Configuring Managed Services

A bundle that needs configuration information should register one or more `ManagedService` objects with a PID service property. If it has a default set of properties for its configuration, it may include them as service properties of the Managed Service. These properties may be used as a configuration template when a `Configuration` object is created for the first time. A Managed Service optionally implements the `MetaTypeProvider` interface to provide information about the property types. See *Meta Typing* on page 46.

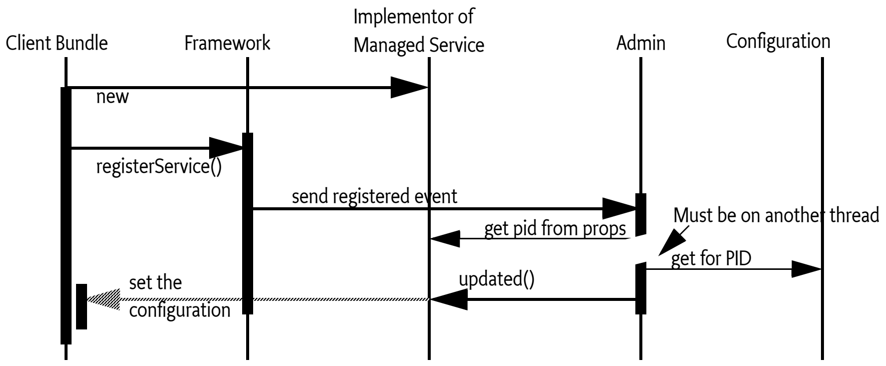
When this registration is detected by the Configuration Admin service, the following steps must occur:

- The configuration stored for the registered PID must be retrieved. If there is a Configuration object for this PID, it is sent to the Managed Service with updated(Dictionary).
- If a Managed Service is registered and no configuration information is available, the Configuration Admin service must call updated(Dictionary) with a null parameter.
- If the Configuration Admin service starts *after* a Managed Service is registered, it must call updated(Dictionary) on this service as soon as possible. For this reason, a Managed Service must always get a callback when it registers *and* the Configuration Admin service is started.

The updated(Dictionary) callback from the Configuration Admin service to the Managed Service must take place asynchronously. This requirement allows the Managed Service to finish its initialization in a synchronized method without interference from the Configuration Admin service callback.

Care should be taken not to cause deadlocks by calling the Framework within a synchronized method.

Figure 11 *Managed Service Configuration Action Diagram*



The updated method may throw a ConfigurationException. This object must describe the problem and what property caused the exception.

3.5.4 Race Conditions

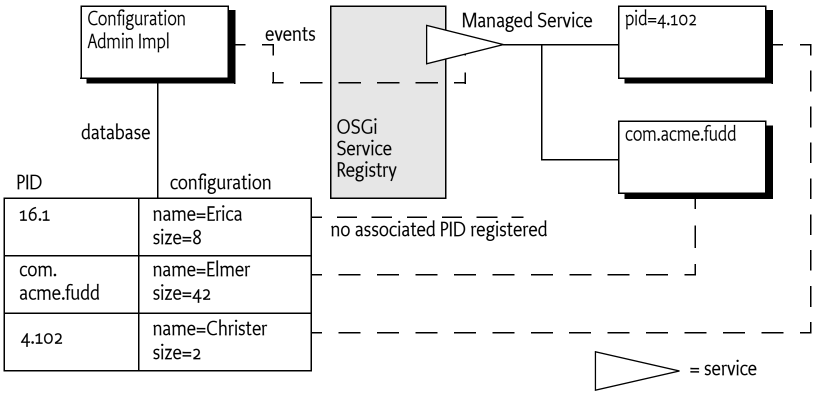
When a Managed Service is registered, the default properties may be visible in the service registry for a short period before they are replaced by the properties of the actual configuration dictionary. Care should be taken that this visibility does not cause race conditions for other bundles.

In cases where race conditions could be harmful, the Managed Service must be split into two pieces: an object performing the actual service and a Managed Service. First, the Managed Service is registered, the configuration is received, and the actual service object is registered. In such cases, the use of a Managed Service Factory that performs this function should be considered.

3.5.5 Examples of Managed Service

Figure 12 shows a Managed Service configuration example. Two services are registered under the ManagedService interface, each with a different PID.

Figure 12 PIDs and External Associations



The Configuration Admin service has a database containing a configuration record for each PID. When the Managed Service with service.pid = com.acme.fudd is registered, the Configuration Admin service will retrieve the properties name=Elmer and size=42 from its database. The properties are stored in a Dictionary object and then given to the Managed Service with the updated(Dictionary) method.

3.5.5.1 Configuring A Console Bundle

In this example, a bundle can run a single debugging console over a Telnet connection. It is a singleton, so it uses a ManagedService object to get its configuration information: the port and the network name on which it should register.

```
class SampleManagedService implements ManagedService {
   Dictionary               properties;
   ServiceRegistration      registration;
   Console                  console;

   public synchronized void start(
      BundleContext context ) throws Exception {
      properties = new Hashtable();
      properties.put( Constants.SERVICE_PID,
         "com.acme.console" );
      properties.put( "port",  new Integer(2011) );

      registration = context.registerService(
         ManagedService.class.getName(),
         this,
         properties
      );
   }
```

```
      public synchronized void updated( Dictionary np ) {
        if ( np != null ) {
          properties = np;
          properties.put(
            Constants.SERVICE_PID, "com.acme.console" );
        }

        if (console == null)
          console = new Console();

        int port = ((Integer)properties.get("port"))
          .intValue();

        String network = (String) properties.get("network");
        console.setPort(port, network);
        registration.setProperties(properties);
      }
      ... further methods
    }
```

3.5.6 Deletion

When a Configuration object for a Managed Service is deleted, the Configuration Admin service must call updated(Dictionary) with a null argument on a thread that is different from that on which the Configuration.delete was executed.

3.6 Managed Service Factory

A Managed Service Factory is used when configuration information is needed for a service that can be instantiated multiple times. When a Managed Service Factory is registered with the Framework, the Configuration Admin service consults its database and calls updated(String,Dictionary) for each associated Configuration object. It passes the identifier of the instance, which can be used as a PID, as well as a Dictionary object with the configuration properties.

A Managed Service Factory is useful when the bundle can provide functionality a number of times, each time with different configuration dictionaries. In this situation, the Managed Service Factory acts like a *class* and the Configuration Admin service can use this Managed Service Factory to *instantiate instances* for that *class*.

In the next section, the word *factory* refers to this concept of creating *instances* of a function defined by a bundle that registers a Managed Service Factory.

3.6.1 When to Use a Managed Service Factory

A Managed Service Factory should be used when a bundle does not have an internal or external entity associated with the configuration information but can potentially be instantiated multiple times.

3.6.1.1 Example Email Fetcher

An email fetcher program displays the number of emails that a user has – a function likely to be required for different users. This function could be viewed as a *class* that needs to be *instantiated* for each user. Each instance requires different parameters, including password, host, protocol, user id, and so on.

An implementation of the Email Fetcher service should register a `ManagedServiceFactory` object. In this way, the Configuration Admin service can define the configuration information for each user separately. The Email Fetcher service will only receive a configuration dictionary for each required instance (user).

3.6.1.2 Example Temperature Conversion Service

Assume a bundle has the code to implement a conversion service that receives a temperature and, depending on settings, can turn an actuator on and off. This service would need to be instantiated many times depending on where it is needed. Each instance would require its own configuration information for the following:

- Upper value
- Lower value
- Switch Identification
- ...

Such a conversion service should register a service object under a `ManagedServiceFactory` interface. A configuration program can then use this Managed Service Factory to create instances as needed. For example, this program could use a Graphic User Interface (GUI) to create such a component and configure it.

3.6.1.3 Serial Ports

Serial ports cannot always be used by the OSGi Device Access specification implementations. Some environments have no means to identify available serial ports, and a device on a serial port cannot always provide information about its type.

Therefore, each serial port requires a description of the device that is connected. The bundle managing the serial ports would need to instantiate a number of serial ports under the control of the Configuration Admin service, with the appropriate `DEVICE_CATEGORY` property to allow it to participate in the Device Access implementation.

If the bundle cannot detect the available serial ports automatically, it should register a Managed Service Factory. The Configuration Admin service can then, with the help of a configuration program, define configuration information for each available serial port.

3.6.2 Registration

Similar to the Managed Service configuration dictionary, the configuration dictionary for a Managed Service Factory is identified by a PID. The Managed Service Factory, however, also has a *factory* PID, which is the PID of the associated Managed Service Factory. It is used to group all Managed Service Factory configuration dictionaries together.

When a Configuration object for a Managed Service Factory is created (ConfigurationAdmin.createFactoryConfiguration), a new unique PID is created for this object by the Configuration Admin service. The scheme used for this PID is defined by the Configuration Admin service and is unrelated to the factory PID.

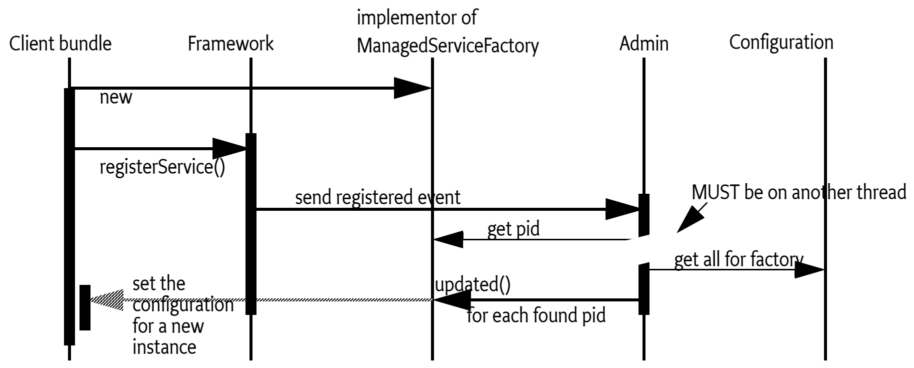
When the Configuration Admin service detects the registration of a Managed Service Factory, it must find all configuration dictionaries for this factory and must then sequentially call ManagedServiceFactory.updated(String,Dictionary) for each configuration dictionary. The first argument is the PID of the Configuration object (the one created by the Configuration Admin service) and the second argument contains the configuration properties.

The Managed Service Factory should then create instances of the associated factory class. Using the PID given in the Configuration object, the bundle may register new services (other than a Managed Service) with the Framework, but this is not required. This may be necessary when the PID is useful in contexts other than the Configuration Admin service.

The receiver must *not* register a Managed Service with this PID because this would force two Configuration objects to have the same PID. If a bundle attempts to do this, the Configuration Admin service should log an error and must ignore the registration of the Managed Service. The configuration dictionary may be used only internally.

The Configuration Admin service must guarantee that the Configuration objects are not deleted before their properties are given to the Managed Service Factory, and must assure that no race conditions exist between initialization and updates.

Figure 13 Managed Service Factory Action Diagram



A Managed Service Factory has only one update method: updated(String, Dictionary). This method can be called any number of times as Configuration objects are created or updated.

The Managed Service Factory must detect whether a PID is being used for the first time, in which case it should create a new *instance*, or a subsequent time, in which case it should update an existing instance.

The Configuration Admin service must call updated(String,Dictionary) on a thread that is different from the one that executed the registration. This requirement allows an implementation of a Managed Service Factory to use a synchronized method to assure that the callbacks do not interfere with the Managed Service Factory registration.

The updated(String,Dictionary) method may throw a ConfigurationException object. This object describes the problem and what property caused the problem. These exceptions should be logged by a Configuration Admin service.

3.6.3 Deletion

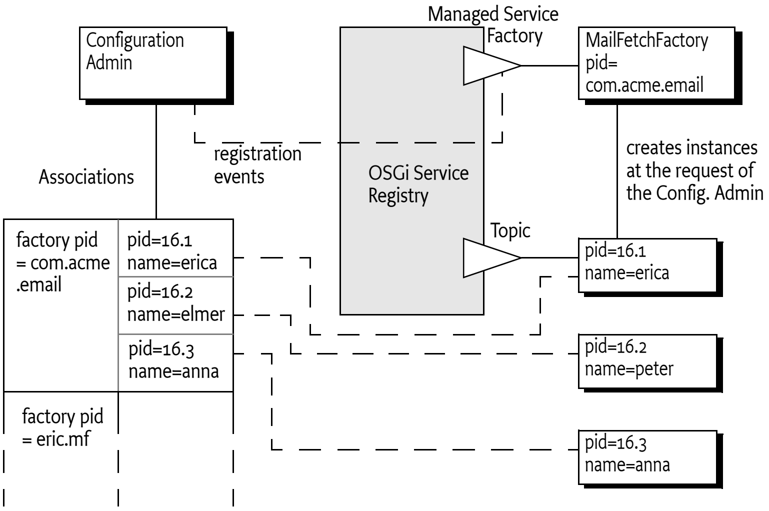
If a configuring bundle deletes an instance of a Managed Service Factory, the deleted(String) method is called. The argument is the PID for this instance. The implementation of the Managed Service Factory must remove all information and stop any behavior associated with that PID. If a service was registered for this PID, it should be unregistered.

3.6.4 Managed Service Factory Example

Figure 14 highlights the differences between a Managed Service and a Managed Service Factory. It shows how a Managed Service Factory implementation receives configuration information that was created before it was registered.

- A bundle implements an EMail Fetcher service. It registers a ManagedServiceFactory object with PID=com.acme.email.
- The Configuration Admin service notices the registration and consults its database. It finds three Configuration objects for which the factory PID is equal to com.acme.email. It must call updated(String,Dictionary) for each of these Configuration objects on the newly registered ManagedServiceFactory object.
- For each configuration dictionary received, the factory should create a new instance of a EMailFetcher object, one for erica (PID=16.1), one for anna (PID=16.3), and one for elmer (PID=16.2).
- The EMailFetcher objects are registered under the Topic interface so their results can be viewed by an online display.
  If the EMailFetcher object is registered, it may safely use the PID of the Configuration object because the Configuration Admin service must guarantee its suitability for this purpose.

Figure 14 *Managed Service Factory Example*



3.6.5 Multiple Consoles Example

This example illustrates how multiple consoles, each of which has its own port and interface can run simultaneously. This approach is very similar to the example for the Managed Service, but highlights the difference by allowing multiple consoles to be created.

```
class ExampleFactory implements ManagedServiceFactory {
  Hashtable consoles = new Hashtable();
  BundleContext context;
  public void start( BundleContext context )
    throws Exception {
    this.context = context;
    Hashtable local = new Hashtable();
    local.put(Constants.SERVICE_PID,"com.acme.console");
    context.registerService(
      ManagedServiceFactory.class.getName(),
      this,
      local );
  }

  public void updated( String pid, Dictionary config ){
    Console console = (Console) consoles.get(pid);
    if (console == null) {
      console = new Console(context);
      consoles.put(pid, console);
    }

    int port = getInt(config, "port", 2011);
    String network = getString(
      config,
      "network",
      null /*all*/
```

```
            );
            console.setPort(port, network);
        }

        public void deleted(String pid) {
            Console console = (Console) consoles.get(pid);
            if (console != null) {
                consoles.remove(pid);
                console.close();
            }
        }
    }
```

3.7 Configuration Admin Service

The ConfigurationAdmin interface provides methods to maintain configuration data in an OSGi environment. This configuration information is defined by a number of Configuration objects associated with specific configuration targets. Configuration objects can be created, listed, modified, and deleted through this interface. Either a remote management system or the bundles configuring their own configuration information may perform these operations.

The ConfigurationAdmin interface has methods for creating and accessing Configuration objects for a Managed Service, as well as methods for managing new Configuration objects for a Managed Service Factory.

3.7.1 Creating a Managed Service Configuration Object

A bundle can create a new Managed Service Configuration object with ConfigurationAdmin.getConfiguration. No create method is offered because doing so could introduce race conditions between different bundles creating the same Configuration object. The getConfiguration method must atomically create and persistently store an object if it does not yet exist.

Two variants of this method are:

- getConfiguration(String) – This method is used by a bundle with a given location to configure its *own* ManagedService objects. The argument specifies the PID of the targeted service.
- getConfiguration(String,String) – This method is used by a management bundle to configure *another* bundle. Therefore, this management bundle needs AdminPermission. The first argument is the PID and the second argument is the location identifier of the targeted ManagedService object.

All Configuration objects have a method, getFactoryPid(), which in this case must return null because the Configuration object is associated with a Managed Service.

Creating a new Configuration object must *not* initiate a callback to the Managed Service updated method.

3.7.2 Creating a Managed Service Factory Configuration Object

The ConfigurationAdmin class provides two methods to create a new instance of a Managed Service Factory:

- createFactoryConfiguration(String) – This method is used by a bundle with a given location to configure its own ManagedServiceFactory objects. The argument specifies the PID of the targeted ManagedServiceFactory object. This *factory PID* can be obtained from the returned Configuration object with the getFactoryPid() method.
- createFactoryConfiguration(String,String)– This method is used by a management bundle to configure another bundle's ManagedServiceFactory object. This management bundle needs AdminPermission. The first argument is the location identifier and the second is the PID of the targeted ManagedServiceFactory object. The *factory PID* can be obtained from the returned Configuration object with getFactoryPid method.

Creating a new factory configuration must *not* initiate a callback to the Managed Service Factory updated method until the properties are set in the Configuration object.

3.7.3 Accessing Existing Configurations

The existing set of Configuration objects can be listed with listConfigurations(String). The argument is a String object with a filter expression. This filter expression has the same syntax as the Framework Filter class. For example:

```
(&(size=42)(service.factoryPid=*osgi*))
```

The filter function must use the properties of the Configuration objects and only return the ones that match the filter expression.

A single Configuration object is identified with a PID and can be obtained with getConfiguration(String).

If the caller has AdminPermission, then all Configuration objects are eligible for search. In other cases, only Configuration objects bound to the calling bundle's location must be returned.

null is returned in both cases when an appropriate Configuration object cannot be found.

3.7.3.1 Updating a Configuration

The process of updating a Configuration object is the same for Managed Services and Managed Service Factories. First, listConfigurations(String) or getConfiguration(String) should be used to get a Configuration object. The properties can be obtained with Configuration.getProperties. When no update has occurred since this object was created, getProperties returns null.

New properties can be set by calling Configuration.update. The Configuration Admin service must first store the configuration information and then call a configuration target's updated method: either the ManagedService.updated or ManagedServiceFactory.updated method. If this target service is not registered, the fresh configuration information must be set when the configuration target service registers.

The update method calls in Configuration objects are not executed synchronously with the related target service updated method. This method must be called asynchronously. The Configuration Admin service, however, must have updated the persistent storage before the update method returns.

3.7.4 Deletion

A Configuration object that is no longer needed can be deleted with Configuration.delete, which removes the Configuration object from the database. The database must be updated before the target service updated method is called.

If the target service is a Managed Service Factory, the factory is informed of the deleted Configuration object by a call to ManagedServiceFactory.deleted. It should then remove the associated *instance*. The ManagedServiceFactory.deleted call must be done asynchronously with respect to Configuration.delete.

When a Configuration object of a Managed Service is deleted, ManagedService.updated is called with null for the properties argument. This method may be used for clean-up, to revert to default values, or to unregister a service.

3.7.5 Updating a Bundle's Own Configuration

The Configuration Admin service specification does not distinguish between updates via a Management Agent and a bundle updating its own configuration information (as defined by its location). Even if a bundle updates its own configuration information, the Configuration Admin service must callback the associated target service updated method.

As a rule, to update its own configuration, a bundle's user interface should *only* update the configuration information and never its internal structures directly. This rule has the advantage that the events, from the bundle implementation's perspective, appear similar for internal updates, remote management updates, and initialization.

3.8 Configuration Plugin

The Configuration Admin service allows third-party applications to participate in the configuration process. Bundles that register a service object under a ConfigurationPlugin interface can process the configuration dictionary just before it reaches the configuration target service.

Plug-ins allow sufficiently privileged bundles to intercept configuration dictionaries just *before* they must be passed to the intended Managed Service or Managed Service Factory but *after* the properties are stored. The changes the plug-in makes are dynamic and must not be stored. The plug-in must only be called when an update takes place while it is registered.

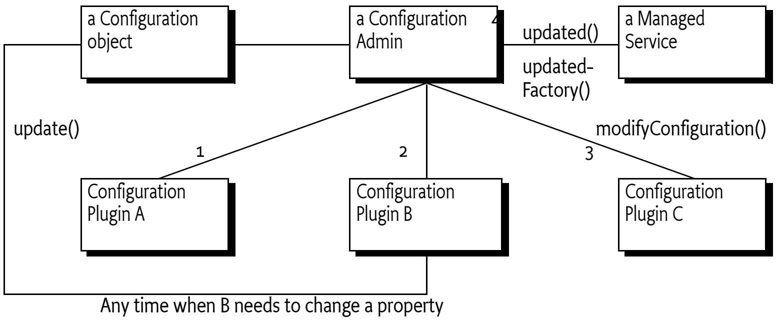
The ConfigurationPlugin interface has only one method: modifyConfiguration(ServiceReference,Dictionary). This method inspects or modifies configuration data.

All plug-ins in the service registry must be traversed and called before the properties are passed to the configuration target service. Each Configuration Plugin object gets a chance to inspect the existing data, look at the target object, which can be a ManagedService object or a ManagedServiceFactory object, and modify the properties of the configuration dictionary. The changes made by a plug-in must be visible to plugins that are called later.

ConfigurationPlugin objects should not modify properties that belong to the configuration properties of the target service unless the implications are understood. This functionality is mainly intended to provide functions that leverage the Framework service registry. The changes made by the plugin should normally not be validated. However, the Configuration Admin must ignore changes to the automatic properties as described in *Automatic Properties* on page 31.

For example, a Configuration Plugin service may add a physical location property to a service. This property can be leveraged by applications that want to know where a service is physically located. This scenario could be carried out without any further support of the service itself, except for the general requirement that the service should propagate the properties it receives from the Configuration Admin service to the service registry.

Figure 15 Order of Configuration Plugin Services



3.8.1 Limiting The Targets

A ConfigurationPlugin object may optionally specify a cm.target registration property. This value is the PID of the configuration target whose configuration updates the ConfigurationPlugin object wants to intercept.

The ConfigurationPlugin object must then only be called with updates for the configuration target service with the specified PID. Omitting the cm.target registration property means that it is called for *all* configuration updates.

3.8.2 Example of Property Expansion

Consider a Managed Service that has a configuration property service.to with the value (objectclass=com.acme.Alarm). When the Configuration Admin service sets this property on the target service, a ConfigurationPlugin object may replace the (objectclass=com.acme.Alarm) filter with an array of existing alarm systems' PIDs as follows:

```
ID "service.to=[32434,232,12421,1212]"
```

A new Alarm Service with service.pid=343 is registered, requiring that the list of the target service be updated. The bundle which registered the Configuration Plugin service, therefore, wants to set the to registration property on the target service. It does *not* do this by calling ManagedService.updated directly for several reasons:

- In a securely configured system, it should not have the permission to make this call or even obtain the target service.
- It could get into race conditions with the Configuration Admin service if it had the permissions in the previous bullet. Both services would compete for access simultaneously.

Instead, it must get the Configuration object from the Configuration Admin service and call the update method on it.

The Configuration Admin service must schedule a new update cycle on another thread, and sometime in the future must call ConfigurationPlugin.modifyProperties. The ConfigurationPlugin object could then set the service.to property to [32434,232,12421,1212, 343]. After that, the Configuration Admin service must call updated on the target service with the new service.to list.

3.8.3 Configuration Data Modifications

Modifications to the configuration dictionary are still under the control of the Configuration Admin service, which must determine whether to accept the changes, hide critical variables, or deny the changes for other reasons.

The ConfigurationPlugin interface must also allow plugins to detect configuration updates to the service via the callback. This ability allows them to synchronize the configuration updates with transient information.

3.8.4 Forcing a Callback

If a bundle needs to force a Configuration Plugin service to be called again, it must fetch the appropriate Configuration object from the Configuration Admin service and call the update() method (the no parameter version) on this object. This call forces an update with the current configuration dictionary so that all applicable plug-ins get called again.

3.8.5 Calling Order

The order in which the ConfigurationPlugin objects are called must depend on the service.cmRanking configuration property of the ConfigurationPlugin object. Table 7 shows the usage of the service.cmRanking property for the order of calling the Configuration Plugin services..

service.cmRanking value	Description
< 0	The Configuration Plugin service should not modify properties and must be called before any modifications are made.
> 0 && <= 1000	The Configuration Plugin service modifies the configuration data. The calling order should be based on the value of the service.cmRanking property.
> 1000	The Configuration Plugin service should not modify data and is called after all modifications are made.

Table 7 service.cmRanking *Usage For Ordering*

3.9 Remote Management

This specification does not attempt to define a remote management interface for the Framework. The purpose of this specification is to define a minimal interface for bundles that is complete enough for testing.

The Configuration Admin service is a primary aspect of remote management, however, and this specification must be compatible with common remote management standards. This section discusses some of the issues of using this specification with [4] *DMTF Common Information Model* (CIM) and [5] *Simple Network Management Protocol* (SNMP), the most likely candidates for remote management today.

These discussions are not complete, comprehensive, or normative. They are intended to point the bundle developer in relevant directions. Further specifications are needed to make a more concrete mapping.

3.9.1 Common Information Model

Common Information Model (CIM) defines the managed objects in [7] *Interface Definition Language* (IDL) language, which was developed for the Common Object Request Broker Architecture (CORBA).

The data types and the data values have a syntax. Additionally, these syntaxes can be mapped to XML. Unfortunately, this XML mapping is very different from the very applicable [6] *XSchema* XML data type definition language. The Framework service registry property types are a proper subset of the CIM data types.

In this specification, a Managed Service Factory maps to a CIM class definition. The primitives `create`, `delete`, and `set` are supported in this specification via the `ManagedServiceFactory` interface. The possible data types in CIM are richer than those the Framework supports and should thus be limited to cases when CIM classes for bundles are defined.

An important conceptual difference between this specification and CIM is the naming of properties. CIM properties are defined within the scope of a class. In this specification, properties are primarily defined within the scope of the Managed Service Factory, but are then placed in the registry, where they have global scope. This mechanism is similar to [8] *Lightweight Directory Access Protocol*, in which the semantics of the properties are defined globally and a class is a collection of globally defined properties.

This specification does not address the non-Configuration Admin service primitives such as notifications and method calls.

3.9.2 Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) defines the data model in ASN.1. SNMP is a rich data typing language that supports many types that are difficult to map to the data types supported in this specification. A large overlap exists, however, and it should be possible to design a data type that is applicable in this context.

The PID of a Managed Service should map to the SNMP Object IDentifier (OID). Managed Service Factories are mapped to tables in SNMP, although this mapping creates an obvious restriction in data types because tables can only contain scalar values. Therefore, the property values of the `Configuration` object would have to be limited to scalar values.

Similar scope issues as seen in CIM arise for SNMP because properties have a global scope in the service registry.

SNMP does not support the concept of method calls or function calls. All information is conveyed as the setting of values. The SNMP paradigm maps closely to this specification.

This specification does not address non-Configuration Admin primitives such as traps.

3.10 Meta Typing

This section discusses how the Metatype specification is used in the context of a Configuration Admin service.

When a Managed Service or Managed Service Factory is registered, the service object may also implement the `MetaTypeProvider` interface.

If the Managed Service or Managed Service Factory object implements the `MetaTypeProvider` interface, a management bundle may assume that the associated `ObjectClassDefinition` object can be used to configure the service.

The `ObjectClassDefinition` and `AttributeDefinition` objects contain sufficient information to automatically build simple user interfaces. They can also be used to augment dedicated interfaces with accurate validations.

When the Metatype specification is used, care should be taken to match the capabilities of the metatype package to the capabilities of the Configuration Admin service specification. Specifically:

- The metatype specification must describe nested arrays and vectors or arrays/vectors of mixed type.

This specification does not address how the metatype is made available to a management system due to the many open issues regarding remote management.

3.11 Security

3.11.1 Permissions

Configuration Admin service security is implemented using ServicePermission and AdminPermission. The following table summarizes the permissions needed by the Configuration Admin bundle itself, as well as those needed by the bundles with which it interacts.

Bundle Registering	ServicePermisson Action	AdminPermission
ConfigurationAdmin	REGISTER ConfigurationAdmin	Yes
	GET ManagedService	
	GET ManagedServiceFactory	
	GET ConfigurationPlugin	
ManagedService	REGISTER ManagedService	No
	GET ConfigurationAdmin	
ManagedServiceFactory	REGISTER ManagedServiceFactory	No
	GET ConfigurationAdmin	
ConfigurationPlugin	REGISTER ConfigurationPlugin	No
	GET ConfigurationAdmin	

Table 8 Permission Overview Configuration Admin

The Configuration Admin service must have ServicePermission[REGISTER, ConfigurationAdmin]. It will also be the only bundle that needs the ServicePermission[GET,ManagedService | ManagedServiceFactory |ConfigurationPlugin]. No other bundle should be allowed to have GET permission for these interfaces. The Configuration Admin bundle must also hold AdminPermission.

Bundles that can be configured must have the ServicePermission[REGISTER,ManagedService|ManagedServiceFactory].

Bundles registering ConfigurationPlugin objects must have the ServicePermission[REGISTER, ConfigurationPlugin]. The Configuration Admin service must trust all services registered with the ConfigurationPlugin interface. Only the Configuration Admin service should have ServicePermission[GET, ConfigurationPlugin.

If a Managed Service or Managed Service Factory is implemented by an object that is also registered under another interface, it is possible, although inappropriate, for a bundle other than the Configuration Admin service implementation to call the updated method. Security-aware bundles can avoid this problem by having their updated methods check that the caller has AdminPermission (such bundles need AdminPermission to perform this check).

Bundles that want to change their own configuration need ServicePermission[GET, ConfigurationAdmin]. A bundle with AdminPermission is allowed to access and modify any Configuration object.

Pre-configuration of bundles requires AdminPermission because the methods that specify a location require this permission.

3.11.2 Forging PIDs

A risk exists of an unauthorized bundle forging a PID in order to obtain and possibly modify the configuration information of another bundle. To mitigate this risk, Configuration objects are generally *bound* to a specific bundle location, and are not passed to any Managed Service or Managed Service Factory registered by a different bundle.

Bundles with the required AdminPermission can create Configuration objects that are not bound. In other words, they have their location set to null. This can be useful for preconfiguring bundles before they are installed without having to know their actual locations.

In this scenario, the Configuration object must become bound to the first bundle that registers a Managed Service (or Managed Service Factory) with the right PID.

A bundle could still possibly obtain another bundle's configuration by registering a Managed Service with the right PID before the victim bundle does so. This situation can be regarded as a denial-of-service attack, because the victim bundle would never receive its configuration information. Such an attack can be avoided by always binding Configuration objects to the right locations. It can also be detected by the Configuration Admin service when the victim bundle registers the correct PID and two equal PIDs are then registered. This violation of this specification should be logged.

3.11.3 Configuration and Permission Administration

Configuration information has a direct influence on the permissions needed by a bundle. For example, when the Configuration Admin Bundle orders a bundle to use port 2011 for a console, that bundle also needs permission for listening to incoming connections on that port.

Both a simple and a complex solution exist for this situation.

The simple solution for this situation provides the bundle with a set of permissions that do not define specific values but allow a range of values. For example, a bundle could listen to ports above 1024 freely. All these ports could then be used for configuration.

The other solution is more complicated. In an environment where there is very strong security, the bundle would only be allowed access to a specific port. This situation requires an atomic update of both the configuration data and the permissions. If this update was not atomic, a potential security hole would exist during the period of time that the set of permissions did not match the configuration.

The following scenario can be used to update a configuration and the security permissions:

1. Stop the bundle.
2. Update the appropriate `Configuration` object via the Configuration Admin service.
3. Update the permissions in the Framework.
4. Start the bundle.

This scenario would achieve atomicity from the point of view of the bundle.

3.12 Configurable Service

Both the Configuration Admin service and the `org.osgi.framework.Configurable` interface address configuration management issues. It is the intention of this specification to replace the Framework interface for configuration management.

The Framework Configurable mechanism works as follows. A registered service object implements the `Configurable` interface to allow a management bundle to configure that service. The `Configurable` interface has only one method: `getConfigurationObject()`. This method returns a Java Bean. Beans can be examined and modified with the `java.reflect` or `java.bean` packages.

This scheme has the following disadvantages:

- *No factory* – Only registered services can be configured, unlike the Managed Service Factory that configures any number of services.
- *Atomicity* – The beans or reflection API can only modify one property at a time and there is no way to tell the bean that no more modifications to the properties will follow. This limitation complicates updates of configurations that have dependencies between properties.
  This specification passes a `Dictionary` object that sets all the configuration properties atomically.
- *Profile* – The Java beans API is linked to many packages that are not likely to be present in OSGi environments. The reflection API may be present but is not simple to use.
  This specification has no required libraries.
- *User Interface support* – UI support in beans is very rudimentary when no AWT is present.
  The associated Metatyping specification does not require any external libraries, and has extensive support for UIs including localization.

3.13 Changes

3.13.1 Clarifications

- It was not clear from the description that a PID received through a Managed Service Factory must not be used to register a Managed Service. This has been highlighted in the appropriate sections.
- It was not clearly specified that a call-back to a target only happens when the data is updated or the target is registered. The creation of a Configuration object does not initiate a call-back. This has been highlighted in the appropriate sections.
- In this release, when a bundle is uninstalled, all Configuration objects that are dynamically bound to that bundle must be unbound again. See *Location Binding* on page 29.
- It was not clearly specified that the data types of a Configuration object allow arrays and vectors that contain elements of mixed types and also null.

3.13.2 Removal of Bundle Location Property

The bundle location property that was required to be set in the Configuration object's properties has been removed because it leaked security sensitive information to all bundles using the Configuration object.

3.13.3 Plug-in Usage

It was not completely clear when a plug-in must be called and how the properties dictionary should behave. This has been clearly specified in *Configuration Plugin* on page 42.

3.13.4 BigInteger/BigDecimal

The classes BigInteger and BigDecimal are not part of the minimal execution requirements and are therefore no longer part of the supported Object types in the Configuration dictionary.

3.13.5 Equals

The behavior of the equals and hashCode methods is now defined. See *Equality* on page 31.

3.13.6 Constant for service.factoryPid

Added a new constant in the ConfigurationAdmin class. See *SERVICE_FACTORYPID* on page 55. This caused this specification to step from version 1.0 to version 1.1.

3.14 org.osgi.service.cm

The OSGi Configuration Admin service Package. Specification Version 1.2

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.cm; specification-version=1.2
```

3.14.1 Summary

- Configuration - The configuration information for a `ManagedService` or `ManagedServiceFactory` object. [p.51]
- ConfigurationAdmin - Service for administering configuration data. [p.54]
- ConfigurationException - An `Exception` class to inform the Configuration Admin service of problems with configuration data. [p.57]
- ConfigurationListener - Listener for Configuration changes. [p.57]
- ConfigurationPlugin - A service interface for processing configuration dictionary before the update. [p.58]
- ManagedService - A service that can receive configuration data from a Configuration Admin service. [p.60]
- ManagedServiceFactory - Manage multiple service instances. [p.62]

3.14.2 public interface Configuration

The configuration information for a `ManagedService` or `ManagedServiceFactory` object. The Configuration Admin service uses this interface to represent the configuration information for a `ManagedService` or for a service instance of a `ManagedServiceFactory`.

A `Configuration` object contains a configuration dictionary and allows the properties to be updated via this object. Bundles wishing to receive configuration dictionaries do not need to use this class - they register a `ManagedService` or `ManagedServiceFactory`. Only administrative bundles, and bundles wishing to update their own configurations need to use this class.

The properties handled in this configuration have case insensitive `String` objects as keys. However, case is preserved from the last set key/value.

A configuration can be *bound* to a bundle location (`Bundle.getLocation()`). The purpose of binding a `Configuration` object to a location is to make it impossible for another bundle to forge a PID that would match this configuration. When a configuration is bound to a specific location, and a bundle with a different location registers a corresponding `ManagedService` object or `ManagedServiceFactory` object, then the configuration is not passed to the updated method of that object.

If a configuration's location is `null`, it is not yet bound to a location. It will become bound to the location of the first bundle that registers a `ManagedService` or `ManagedServiceFactory` object with the corresponding PID.

The same `Configuration` object is used for configuring both a Managed Service Factory and a Managed Service. When it is important to differentiate between these two the term “factory configuration” is used.

3.14.2.1 public void delete( ) throws IOException

☐ Delete this `Configuration` object. Removes this configuration object from the persistent store. Notify asynchronously the corresponding Managed Service or Managed Service Factory. A `ManagedService` object is notified by a call to its `updated` method with a `null` properties argument. A `ManagedServiceFactory` object is notified by a call to its `deleted` method. Also intiates a call to any `ConfigurationListeners` asynchronously.

Throws `IOException` – If delete fails

`IllegalStateException` – if this configuration has been deleted

3.14.2.2 public boolean equals(Object other)

other `Configuration` object to compare against

☐ Equality is defined to have equal PIDs Two Configuration objects are equal when their PIDs are equal.

Returns `true` if equal, `false` if not a `Configuration` object or one with a different PID.

3.14.2.3 public String getBundleLocation( )

☐ Get the bundle location. Returns the bundle location to which this configuration is bound, or `null` if it is not yet bound to a bundle location.

This call requires `AdminPermission`.

Returns location to which this configuration is bound, or `null`.

Throws `SecurityException` – if the caller does not have `AdminPermission`.

`IllegalStateException` – if this `Configuration` object has been deleted.

3.14.2.4 public String getFactoryPid( )

☐ For a factory configuration return the PID of the corresponding Managed Service Factory, else return `null`.

Returns factory PID or `null`

Throws `IllegalStateException` – if this configuration has been deleted

3.14.2.5 public String getPid( )

☐ Get the PID for this `Configuration` object.

Returns the PID for this `Configuration` object.

Throws `IllegalStateException` – if this configuration has been deleted

3.14.2.6 public Dictionary getProperties( )

☐ Return the properties of this `Configuration` object. The `Dictionary` object returned is a private copy for the caller and may be changed without influencing the stored configuration. The keys in the returned dictionary are case insensitive and are always of type `String`.

If called just after the configuration is created and before update has been called, this method returns `null`.

Returns A private copy of the properties for the caller or `null`. These properties must not contain the "service.bundleLocation" property. The value of this property may be obtained from the `getBundleLocation` method.

Throws `IllegalStateException` – if this configuration has been deleted

3.14.2.7 public int hashCode( )

□ Hash code is based on PID. The hashcode for two Configuration objects must be the same when the Configuration PID's are the same.

Returns hash code for this Configuration object

3.14.2.8 public void setBundleLocation(String bundleLocation)

bundleLocation a bundle location or `null`

□ Bind this `Configuration` object to the specified bundle location. If the bundleLocation parameter is `null` then the `Configuration` object will not be bound to a location. It will be set to the bundle's location before the first time a Managed Service/Managed Service Factory receives this `Configuration` object via the updated method and before any plugins are called. The bundle location will be set persistently.

This method requires `AdminPermission`.

Throws `SecurityException` – if the caller does not have `AdminPermission`

`IllegalStateException` – if this configuration has been deleted

3.14.2.9 public void update(Dictionary properties) throws IOException

properties the new set of properties for this configuration

□ Update the properties of this `Configuration` object. Stores the properties in persistent storage after adding or overwriting the following properties:

- "service.pid" : is set to be the PID of this configuration.
- "service.factoryPid" : if this is a factory configuration it is set to the factory PID else it is not set.

These system properties are all of type `String`.

If the corresponding Managed Service/Managed Service Factory is registered, its updated method must be called asynchronously. Else, this callback is delayed until aforementioned registration occurs. Also intiates a call to any `ConfigurationListeners` asynchronously.

Throws `IOException` – if update cannot be made persistent

`IllegalArgumentException` – if the `Dictionary` object contains invalid configuration types or contains case variants of the same key name.

`IllegalStateException` – if this configuration has been deleted

3.14.2.10 public void update( ) throws IOException

□ Update the `Configuration` object with the current properties. Initiate the `updated` callback to the Managed Service or Managed Service Factory with the current properties asynchronously. Also intiates a call to any `ConfigurationListeners` asynchronously.

This is the only way for a bundle that uses a Configuration Plugin service to initate a callback. For example, when that bundle detects a change that requires an update of the Managed Service or Managed Service Factory via its `ConfigurationPlugin` object.

Throws `IOException` – if update cannot access the properties in persistent storage

`IllegalStateException` – if this configuration has been deleted

See Also `ConfigurationPlugin`[p.58]

3.14.3 public interface ConfigurationAdmin

Service for administering configuration data.

The main purpose of this interface is to store bundle configuration data persistently. This information is represented in `Configuration` objects. The actual configuration data is a `Dictionary` of properties inside a `Configuration` object.

There are two principally different ways to manage configurations. First there is the concept of a Managed Service, where configuration data is uniquely associated with an object registered with the service registry.

Next, there is the concept of a factory where the Configuration Admin service will maintain 0 or more `Configuration` objects for a Managed Service Factory that is registered with the Framework.

The first concept is intended for configuration data about "things/services" whose existence is defined externally, e.g. a specific printer. Factories are intended for "things/services" that can be created any number of times, e.g. a configuration for a DHCP server for different networks.

Bundles that require configuration should register a Managed Service or a Managed Service Factory in the service registry. A registration property named `service.pid` (persistent identifier or PID) must be used to identify this Managed Service or Managed Service Factory to the Configuration Admin service.

When the ConfigurationAdmin detects the registration of a Managed Service, it checks its persistent storage for a configuration object whose PID matches the PID registration property (`service.pid`) of the Managed Service. If found, it calls `ManagedService.updated`[p.61] method with the new properties. The implementation of a Configuration Admin service must run these call-backs asynchronously to allow proper synchronization.

When the Configuration Admin service detects a Managed Service Factory registration, it checks its storage for configuration objects whose `factoryPid` matches the PID of the Managed Service Factory. For each such `Configuration` objects, it calls the `ManagedServiceFactory.updated` method asynchronously with the new properties. The calls to the `updated` method of a `ManagedServiceFactory` must be executed sequentially and not overlap in time.

In general, bundles having permission to use the Configuration Admin service can only access and modify their own configuration information. Accessing or modifying the configuration of another bundle requires `AdminPermission`.

`Configuration` objects can be *bound* to a specified bundle location. In this case, if a matching Managed Service or Managed Service Factory is registered by a bundle with a different location, then the Configuration Admin service must not do the normal callback, and it should log an error. In the case where a `Configuration` object is not bound, its location field is `null`, the Configuration Admin service will bind it to the location of the bundle that registers the first Managed Service or Managed Service Factory that has a corresponding PID property. When a `Configuration` object is bound to a bundle location in this manner, the Confguration Admin service must detect if the bundle corresponding to the location is uninstalled. If this occurs, the `Configuration` object is unbound, that is its location field is set back to `null`.

The method descriptions of this class refer to a concept of "the calling bundle". This is a loose way of referring to the bundle which obtained the Configuration Admin service from the service registry. Implementations of `ConfigurationAdmin` must use a `org.osgi.framework.ServiceFactory` to support this concept.

3.14.3.1 public static final String SERVICE_BUNDLELOCATION = "service.bundleLocation"

Service property naming the location of the bundle that is associated with a a `Configuration` object. This property can be searched for but must not appear in the configuration dictionary for security reason. The property's value is of type `String`.

Since 1.1

3.14.3.2 public static final String SERVICE_FACTORYPID = "service.factoryPid"

Service property naming the Factory PID in the configuration dictionary. The property's value is of type `String`.

Since 1.1

3.14.3.3 public Configuration createFactoryConfiguration(String factoryPid) throws IOException

factoryPid PID of factory (not `null`).

□ Create a new factory `Configuration` object with a new PID. The properties of the new `Configuration` object are `null` until the first time that its `Configuration.update(Dictionary)`[p.53] method is called.

It is not required that the `factoryPid` maps to a registered Managed Service Factory.

The `Configuration` object is bound to the location of the calling bundle.

Returns a new `Configuration` object.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if caller does not have `AdminPermission` and `factoryPid` is bound to another bundle.

3.14.3.4 public Configuration createFactoryConfiguration(String factoryPid, String location) throws IOException

factoryPid PID of factory (not `null`).

location a bundle location string, or `null`.

□ Create a new factory `Configuration` object with a new PID. The properties of the new `Configuration` object are `null` until the first time that its `Configuration.update(Dictionary)`[p.53] method is called.

It is not required that the `factoryPid` maps to a registered Managed Service Factory.

The `Configuration` is bound to the location specified. If this location is `null` it will be bound to the location of the first bundle that registers a Managed Service Factory with a corresponding PID.

This method requires `AdminPermission`.

Returns a new `Configuration` object.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if caller does not have `AdminPermission`.

3.14.3.5 public Configuration getConfiguration(String pid, String location) throws IOException

pid persistent identifier.

location the bundle location string, or `null`.

☐ Get an existing `Configuration` object from the persistent store, or create a new `Configuration` object.

If a `Configuration` with this PID already exists in Configuration Admin service return it. The location parameter is ignored in this case.

Else, return a new `Configuration` object. This new object is bound to the location and the properties are set to `null`. If the location parameter is `null`, it will be set when a Managed Service with the corresponding PID is registered for the first time.

This method requires `AdminPermission`.

Returns an existing or new `Configuration` object.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if the caller does not have `AdminPermission`.

3.14.3.6 public Configuration getConfiguration(String pid) throws IOException

pid persistent identifier.

☐ Get an existing or new `Configuration` object from the persistent store. If the `Configuration` object for this PID does not exist, create a new `Configuration` object for that PID, where properties are `null`. Bind its location to the calling bundle's location.

Else, if the location of the existing `Configuration` object is `null`, set it to the calling bundle's location.

If the location of the `Configuration` object does not match the calling bundle, throw a `SecurityException`.

Returns an existing or new `Configuration` matching the PID.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if the `Configuration` object is bound to a location different from that of the calling bundle and it has no `AdminPermission`.

3.14.3.7 public Configuration[] listConfigurations(String filter) throws

IOException, InvalidSyntaxException

filter a `Filter` object, or `null` to retrieve all `Configuration` objects.

□ List the current `Configuration` objects which match the filter.

Only `Configuration` objects with non- `null` properties are considered current. That is, `Configuration.getProperties()` is guaranteed not to return `null` for each of the returned `Configuration` objects.

Normally only `Configuration` objects that are bound to the location of the calling bundle are returned. If the caller has `AdminPermission`, then all matching `Configuration` objects are returned.

The syntax of the filter string is as defined in the `Filter` class. The filter can test any configuration parameters including the following system properties:

- `service.pid`-`String`- the PID under which this is registered
- `service.factoryPid`-`String`- the factory if applicable
- `service.bundleLocation`-`String`- the bundle location

The filter can also be `null`, meaning that all `Configuration` objects should be returned.

Returns all matching `Configuration` objects, or `null` if there aren't any

Throws `IOException` – if access to persistent storage fails

`InvalidSyntaxException` – if the filter string is invalid

3.14.4 public class ConfigurationException extends Exception

An `Exception` class to inform the Configuration Admin service of problems with configuration data.

3.14.4.1 public ConfigurationException(String property, String reason)

property name of the property that caused the problem, `null` if no specific property was the cause

reason reason for failure

□ Create a `ConfigurationException` object.

3.14.4.2 public String getProperty( )

□ Return the property name that caused the failure or null.

Returns name of property or null if no specific property caused the problem

3.14.4.3 public String getReason( )

□ Return the reason for this exception.

Returns reason of the failure

3.14.5 public interface ConfigurationListener

Listener for Configuration changes.

`ConfigurationListener` objects are registered with the Framework service registry and are notified when a `Configuration` object is updated or deleted.

`ConfigurationListener` objects are passed the type of configuration change.

One of the change methods will be called with `CM_UPDATED` when `Configuration.update` is called or with `CM_DELETED` when `Configuration.delete` is called. Notification will be asynchronous to the update or delete method call. The design is very lightweight in that is does not pass `Configuration` objects, the listener is merely advised that the configuration information for a given pid has changed. If the listener wants to locate the `Configuration` object for the specified pid, it must use `ConfigurationAdmin`.

Security Considerations. Bundles wishing to monitor `Configuration` changes will require `ServicePermission[ConfigurationListener, REGISTER]` to register a `ConfigurationListener` service. Since `Configuration` objects are not passed to the listener, no sensitive configuration information is available to the listener.

3.14.5.1 public static final int CM_DELETED = 2

Change type that indicates that `Configuration.delete` was called.

3.14.5.2 public static final int CM_UPDATED = 1

Change type that indicates that `Configuration.update` was called.

3.14.5.3 public void configurationChanged(String pid, int type)

pid The pid of the configuration which changed.

type The type of the configuration change.

□ Receives notification a configuration has changed.

This method is only called if the target of the configuration is a ManagedService.

3.14.5.4 public void factoryConfigurationChanged(String factoryPid, String pid, int type)

factoryPid The factory pid for the changed configuration.

pid The pid of the configuration which changed.

type The type of the configuration change.

□ Receives notification a factory configuration has changed.

This method is only called if the target of the configuration is a ManagedServiceFactory.

3.14.6 public interface ConfigurationPlugin

A service interface for processing configuration dictionary before the update.

A bundle registers a `ConfigurationPlugin` object in order to process configuration updates before they reach the Managed Service or Managed Service Factory. The Configuration Admin service will detect registrations of Configuration Plugin services and must call these services every time before it calls the `ManagedService` or `ManagedServiceFactoryupdated` method. The Configuration Plugin service thus has the opportunity to view and modify the properties before they are passed to the ManagedS ervice or Managed Service Factory.

Configuration Plugin (plugin) services have full read/write access to all configuration information. Therefore, bundles using this facility should be trusted. Access to this facility should be limited with `ServicePermission[REGISTER, ConfigurationPlugin]`. Implementations of a Configuration Plugin service should assure that they only act on appropriate configurations.

The `Integerservice.cmRanking` registration property may be specified. Not specifying this registration property, or setting it to something other than an `Integer`, is the same as setting it to the `Integer` zero. The `service.cmRanking` property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones. In the event of more than one plugin having the same value of `service.cmRanking`, then the Configuration Admin service arbitrarily chooses the order in which they are called.

By convention, plugins with `service.cmRanking< 0` or `service.cmRanking >1000` should not make modifications to the properties.

The Configuration Admin service has the right to hide properties from plugins, or to ignore some or all the changes that they make. This might be done for security reasons. Any such behavior is entirely implementation defined.

A plugin may optionally specify a `cm.target` registration property whose value is the PID of the Managed Service or Managed Service Factory whose configuration updates the plugin is intended to intercept. The plugin will then only be called with configuration updates that are targetted at the Managed Service or Managed Service Factory with the specified PID. Omitting the `cm.target` registration property means that the plugin is called for all configuration updates.

3.14.6.1 public static final String CM_RANKING = "service.cmRanking"

A service property to specify the order in which plugins are invoked. This property contains an `Integer` ranking of the plugin. Not specifying this registration property, or setting it to something other than an `Integer`, is the same as setting it to the `Integer` zero. This property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones.

Since 1.2

3.14.6.2 public static final String CM_TARGET = "cm.target"

A service property to limit the Managed Service or Managed Service Factory configuration dictionaries a Configuration Plugin service receives. This property contains a `String[]` of PIDs. A Configuration Admin service must call a Configuration Plugin service only when this property is not set, or the target service's PID is listed in this property.

3.14.6.3 public void modifyConfiguration(ServiceReference reference, Dictionary properties)

reference reference to the Managed Service or Managed Service Factory

properties The configuration properties. This argument must not contain the "service.bundleLocation" property. The value of this property may be obtained from the `Configuration.getBundleLocation` method.

□ View and possibly modify the a set of configuration properties before they are sent to the Managed Service or the Managed Service Factory. The Configuration Plugin services are called in increasing order of their `service.cmRanking` property. If this property is undefined or is a non-`Integer` type, 0 is used.

This method should not modify the properties unless the `service.cmRanking` of this plugin is in the range `0 <= service.cmRanking <= 1000`.

If this method throws any `Exception`, the Configuration Admin service must catch it and should log it.

3.14.7 public interface ManagedService

A service that can receive configuration data from a Configuration Admin service.

A Managed Service is a service that needs configuration data. Such an object should be registered with the Framework registry with the `service.pid` property set to some unique identitifier called a PID.

If the Configuration Admin service has a `Configuration` object corresponding to this PID, it will callback the `updated()` method of the `ManagedService` object, passing the properties of that `Configuration` object.

If it has no such `Configuration` object, then it calls back with a `null` properties argument. Registering a Managed Service will always result in a callback to the `updated()` method provided the Configuration Admin service is, or becomes active. This callback must always be done asynchronously.

Else, every time that either of the `updated()` methods is called on that `Configuration` object, the `ManagedService.updated()` method with the new properties is called. If the `delete()` method is called on that `Configuration` object, `ManagedService.updated()` is called with a `null` for the properties parameter. All these callbacks must be done asynchronously.

The following example shows the code of a serial port that will create a port depending on configuration information.

```
class SerialPort implements ManagedService {

  ServiceRegistration registration;
  Hashtable configuration;
  CommPortIdentifier id;

  synchronized void open(CommPortIdentifier id,
  BundleContext context) {
    this.id = id;
    registration = context.registerService(
      ManagedService.class.getName(),
      this,
      null // Properties will come from CM in updated
    );
  }

  Hashtable getDefaults() {
    Hashtable defaults = new Hashtable();
    defaults.put( “port”, id.getName() );
    defaults.put( “product”, “unknown” );
    defaults.put( “baud”, “9600” );
    defaults.put( Constants.SERVICE_PID,
      “com.acme.serialport.” + id.getName() );
    return defaults;
  }

  public synchronized void updated(
    Dictionary configuration  ) {
    if ( configuration ==
null
)
      registration.setProperties( getDefaults() );
    else {
      setSpeed( configuration.get(”baud”) );
      registration.setProperties( configuration );
    }
  }
  ...
}
```

As a convention, it is recommended that when a Managed Service is updated, it should copy all the properties it does not recognize into the service registration properties. This will allow the Configuration Admin service to set properties on services which can then be used by other applications.

3.14.7.1 public void updated(Dictionary properties) throws ConfigurationException

properties A copy of the Configuration properties, or `null`. This argument must not contain the “service.bundleLocation” property. The value of this property may be obtained from the `Configuration.getBundleLocation` method.

- Update the configuration for a Managed Service.

When the implementation of updated(Dictionary) detects any kind of error in the configuration properties, it should create a new ConfigurationException which describes the problem. This can allow a management system to provide useful information to a human administrator.

If this method throws any other Exception, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously which initiated the callback. This implies that implementors of Managed Service can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

Throws ConfigurationException – when the update fails

3.14.8 public interface ManagedServiceFactory

Manage multiple service instances. Bundles registering this interface are giving the Configuration Admin service the ability to create and configure a number of instances of a service that the implementing bundle can provide. For example, a bundle implementing a DHCP server could be instantiated multiple times for different interfaces using a factory.

Each of these *service instances* is represented, in the persistent storage of the Configuration Admin service, by a factory Configuration object that has a PID. When such a Configuration is updated, the Configuration Admin service calls the ManagedServiceFactory updated method with the new properties. When updated is called with a new PID, the Managed Service Factory should create a new factory instance based on these configuration properties. When called with a PID that it has seen before, it should update that existing service instance with the new configuration information.

In general it is expected that the implementation of this interface will maintain a data structure that maps PIDs to the factory instances that it has created. The semantics of a factory instance are defined by the Managed Service Factory. However, if the factory instance is registered as a service object with the service registry, its PID should match the PID of the corresponding Configuration object (but it should **not** be registered as a Managed Service!).

An example that demonstrates the use of a factory. It will create serial ports under command of the Configuration Admin service.

```
class SerialPortFactory
  implements ManagedServiceFactory {
  ServiceRegistration registration;
  Hashtable ports;
  void start(BundleContext context) {
    Hashtable properties = new Hashtable();
    properties.put( Constants.SERVICE_PID,
      “com.acme.serialportfactory” );
    registration = context.registerService(
      ManagedServiceFactory.class.getName(),
        this,
        properties
      );
    }
    public void updated( String pid,
      Dictionary properties  ) {
      String portName = (String) properties.get("port");
      SerialPortService port =
        (SerialPort) ports.get( pid );
      if ( port == null ) {
        port = new SerialPortService();
        ports.put( pid, port );
        port.open();
      }
      if ( port.getPortName().equals(portName) )
        return;
      port.setPortName( portName );
    }
    public void deleted( String pid ) {
      SerialPortService port =
        (SerialPort) ports.get( pid );
      port.close();
      ports.remove( pid );
    }
    ...
  }
```

3.14.8.1 public void deleted(String pid)

pid the PID of the service to be removed

□ Remove a factory instance. Remove the factory instance associated with the PID. If the instance was registered with the service registry, it should be unregistered.

If this method throws any `Exception`, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously.

3.14.8.2 public String getName( )

□ Return a descriptive name of this factory.

Returns the name for the factory, which might be localized

3.14.8.3 public void updated(String pid, Dictionary properties) throws ConfigurationException

pid The PID for this configuration.

properties A copy of the configuration properties. This argument must not contain the service.bundleLocation" property. The value of this property may be obtained from the `Configuration. getBundleLocation` method.

- Create a new instance, or update the configuration of an existing instance. If the PID of the `Configuration` object is new for the Managed Service Factory, then create a new factory instance, using the configuration `properties` provided. Else, update the service instance with the provided `properties`.

  If the factory instance is registered with the Framework, then the configuration `properties` should be copied to its registry properties. This is not mandatory and security sensitive properties should obviously not be copied.

  If this method throws any `Exception`, the Configuration Admin service must catch it and should log it.

  When the implementation of updated detects any kind of error in the configuration properties, it should create a new `ConfigurationException`[p.57] which describes the problem.

  The Configuration Admin service must call this method asynchronously. This implies that implementors of the `ManagedServiceFactory` class can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

Throws `ConfigurationException` – when the configuration properties are invalid.

3.15 References

[4] *DMTF Common Information Model*
http://www.dmtf.org

[5] *Simple Network Management Protocol*
RFCs http://directory.google.com/Top/Computers/Internet/Protocols/SNMP/RFCs

[6] *XSchema*
http://www.w3.org/TR/xmlschema-0/

[7] *Interface Definition Language*
http://www.omg.org

[8] *Lightweight Directory Access Protocol*
http://directory.google.com/Top/Computers/Software/Internet/Servers/Directory/LDAP

[9] *Understanding and Deploying LDAP Directory services*
Timothy Howes et. al. ISBN 1-57870-070-1, MacMillan Technical publishing.

4 Metatype Specification

Version 1.0

4.1 Introduction

The Metatype specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using so-called *metadata*.

The purpose of this specification is to allow services to specify the type information of data that they can use as arguments. The data is based on *attributes*, which are key/value pairs like properties.

A designer in a type-safe language like Java is often confronted with the choice of using the language constructs to exchange data or using a technique based on attributes/properties that are based on key/value pairs. Attributes provide an escape from the rigid type-safety requirements of modern programming languages.

Type-safety works very well for software development environments in which multiple programmers work together on large applications or systems, but often lacks the flexibility needed to receive structured data from the outside world.

The attribute paradigm has several characteristics that make this approach suitable when data needs to be communicated between different entities which “speak” different languages. Attributes are uncomplicated, resilient to change, and allow the receiver to dynamically adapt to different types of data.

As an example, the OSGi Service Platform Specifications define several attribute types which are used in a Framework implementation, but which are also used and referenced by other OSGi specifications such as the *Configuration Admin Service Specification* on page 23. A Configuration Admin service implementation deploys attributes (key/value pairs) as configuration properties.

During the development of the Configuration Admin service, it became clear that the Framework attribute types needed to be described in a computer readable form. This information (the metadata) could then be used to automatically create user interfaces for management systems or could be translated into management information specifications such as CIM, SNMP, and the like.

4.1.1 Essentials

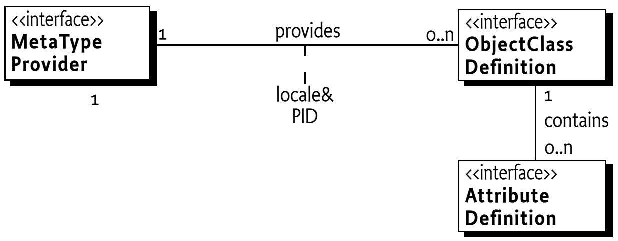
- *Conceptual model* – The specification must have a conceptual model for how classes and attributes are organized.

- *Standards* – The specification should be aligned with appropriate standards, and explained in situations where the specification is not aligned with, or cannot be mapped to, standards.
- *Remote Management* – Remote management should be taken into account.
- *Size* – Minimal overhead in size for a bundle using this specification is required.
- *Localization* – It must be possible to use this specification with different languages at the same time. This ability allows servlets to serve information in the language selected in the browser.
- *Type information* – The definition of an attribution should contain the name (if it is required), the cardinality, a label, a description, labels for enumerated values, and the Java class that should be used for the values.
- *Validation* – It should be possible to validate the values of the attributes.

4.1.2 Entities

- *Attribute* – A key/value pair.
- *AttributeDefinition* – Defines a description, name, help text, and type information of an attribute.
- *ObjectClassDefinition* – Defines the type of a datum. It contains a description and name of the type plus a set of AttributeDefinition objects.
- *MetaTypeProvider* – Provides access to the object classes that are available for this object. Access uses the PID and a locale to find the best ObjectClassDefinition object.

Figure 16 *Class Diagram Meta Typing, org.osgi.service.metatyping*



4.1.3 Operation

This specification starts with an object that implements the MetaTypeProvider interface. It is not specified how this object is obtained, and there are several possibilities. Often, however, this object is a service registered with the Framework.

A MetaTypeProvider object provides access to ObjectClassDefinition objects. These objects define all the information for a specific *object class*. An object class is a some descriptive information and a set of named attributes (which are key/value pairs).

Access to object classes is qualified by a locale and a Persistent IDentity (PID). The locale is a String object that defines for which language the ObjectClassDefinition is intended, allowing for localized user interfaces. The PID is used when a single MetaTypeProvider object can provide ObjectClassDefinition objects for multiple purposes. The context in which the MetaTypeProvider object is used should make this clear.

Attributes have global scope. Two object classes can consist of the same attributes, and attributes with the same name should have the same definition. This global scope is unlike languages like Java that scope instance variables within a class, but it is similar to the Lightweight Directory Access Protocol (LDAP) (SNMP also uses a global attribute name-space).

Attribute Definition objects provide sufficient localized information to generate user interfaces.

4.2 Attributes Model

The Framework uses the LDAP filter syntax for searching the Framework registry. The usage of the attributes in this specification and the Framework specification closely resemble the LDAP attribute model. Therefore, the names used in this specification have been aligned with LDAP. Consequently, the interfaces which are defined by this Specification are:

- AttributeDefinition
- ObjectClassDefinition
- MetaTypeProvider

These names correspond to the LDAP attribute model. For further information on ASN.1-defined attributes and X.500 object classes and attributes, see [11] *Understanding and Deploying LDAP Directory services*.

The LDAP attribute model assumes a global name-space for attributes, and object classes consist of a number of attributes. So, if an object class inherits the same attribute from different parents, only one copy of the attribute must become part of the object class definition. This name-space implies that a given attribute, for example cn, should *always* be the common name and the type must always be a String. An attribute cn cannot be an Integer in another object class definition. In this respect, the OSGi approach towards attribute definitions is comparable with the LDAP attribute model.

4.3 Object Class Definition

The ObjectClassDefinition interface is used to group the attributes which are defined in AttributeDefinition objects.

An ObjectClassDefinition object contains the information about the overall set of attributes and has the following elements:

- A name which can be returned in different locales.
- A global name-space in the registry, which is the same condition as LDAP/X.500 object classes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organizations, and many

companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned. This id can be a Java class name (reverse domain name) or can be generated with a GUID algorithm. All LDAP-defined object classes already have an associated OID. It is strongly advised to define the object classes from existing LDAP schemes which provide many preexisting OIDs. Many such schemes exist ranging from postal addresses to DHCP parameters.
- A human-readable description of the class.
- A list of attribute definitions which can be filtered as required, or optional. Note that in X.500 the mandatory or required status of an attribute is part of the object class definition and not of the attribute definition.
- An icon, in different sizes.

4.4 Attribute Definition

The `AttributeDefinition` interface provides the means to describe the data type of attributes.

The `AttributeDefinition` interface defines the following elements:

- Defined names (final ints) for the data types as restricted in the Framework for the attributes, called the syntax in OSI terms, which can be obtained with the `getType()` method.
- `AttributeDefinition` objects should use and ID that is similar to the OID as described in the ID field for `ObjectClassDefinition`.
- A localized name intended to be used in user interfaces.
- A localized description that defines the semantics of the attribute and possible constraints, which should be usable for tooltips.
- An indication if this attribute should be stored as a unique value, a `Vector`, or an array of values, as well as the maximum cardinality of the type.
- The data type, as limited by the Framework service registry attribute types.
- A validation function to verify if a possible value is correct.
- A list of values and a list of localized labels. Intended for popup menus in GUIs, allowing the user to choose from a set.
- A default value. The return type of this is a `String[ ]`. For cardinality = zero, this return type must be an array of one `String` object. For other cardinalities, the array must not contain more than the absolute value of *cardinality* `String` objects. In that case, it may contain 0 objects.

4.5 Meta Type Provider

The `MetaTypeProvider` interface is used to access metatype information. It is used in management systems and run-time management. It supports locales so that the text used in `AttributeDefinition` and `ObjectClassDefinition` objects can be adapted to different locales.

The PID is given as an argument with the `getObjectClassDefinition` method so that a single `MetaTypeProvider` object can be used for different object classes with their own PIDs.

Locale objects are represented in String objects because not all profiles support Locale. The String holds the standard Locale presentation of:

```
<language> [ "_" <country> [ "_" <variation>]]
```

For example, "en", "nl_BE", "en_CA_posix".

4.6 Metatype Example

AttributeDefinition and ObjectClassDefinition classes are intended to be easy to use for bundles. This example shows a naive implementation for these classes (note that the get methods usages are not shown). Commercial implementations can use XML, Java serialization, or Java Properties for implementations. This example uses plain code to store the definitions.

The example first shows that the ObjectClassDefinition interface is implemented in the OCD class. The name is made very short because the class is used to instantiate the static structures. Normally many of these objects are instantiated very close to each other, and long names would make these lists of instantiations very long.

```
class OCD implements ObjectClassDefinition {
   String                    name;
   String                    id;
   String                    description;
   AttributeDefinition       required[];
   AttributeDefinition       optional[];

   public OCD(
      String name, String id, String description,
      AttributeDefinition required[],
      AttributeDefinition optional[]) {

      this.name = name;
      this.id = id;
      this.description = description;
      this.required = required;
      this.optional = optional;
   }
   .... All the get methods
}
```

The second class is the AD class that implements the AttributeDefinition interface. The name is short for the same reason as in OCD. Note the two different constructors to simplify the common case.

```
class AD implements AttributeDefinition {
   String                    name;
   String                    id;
   String                    description;
   int                       cardinality;
   int                       syntax;
```

```
    String[]                    values;
    String[]                    labels;
    String[]                    deflt;

    public AD( String name, String id, String description,
      int syntax, int cardinality, String values[],
      String labels[], String deflt[]) {
        this.name               = name;
        this.id                 = id;
        this.description        = description;
        this.cardinality        = cardinality;
        this.syntax             = syntax;
        this.values             = values;
        this.labels             = labels;
    }

    public AD( String name, String id, String description,
      int syntax)
    {
      this(name,id,description,syntax,0,null,null, null);
    }
    ... All the get methods and validate method
  }
```

The last part is the example that implements a MetaTypeProvider class. Only one locale is supported, the US locale. The OIDs used in this example are the actual OIDs as defined in X.500.

```
  public class Example implements MetaTypeProvider {
    final static AD cn = new AD(
      "cn",         "2.5.4.3", "Common name", AD.STRING);
    final static AD sn = new AD(
      "sn",         "2.5.4.4", "Sur name", AD.STRING);
    final static AD description = new AD(
      "description", "2.5.4.13","Description", AD.STRING);
    final static AD seeAlso = new AD(
      "seeAlso",    "2.5.4.34", "See Also", AD.STRING);
    final static AD telephoneNumber = new AD(
      "telephoneNumber", "2.5.4.20", "Tel nr", AD.STRING);
    final static AD userPassword = new AD(
      "userPassword", "2.5.4.3", "Password", AD.STRING);

    final static ObjectClassDefinition person = new OCD(
      "person", "2.5.6.6", "Defines a person",
        new AD[] { cn, sn },
        new AD[] { description, seeAlso,
          telephoneNumber, userPassword}
    );

    public ObjectClassDefinition getObjectClassDefinition(
      String pid, String locale) {
      return person;
    }
    public String[] getLocales() {
      return new String[] { "en_US" };
    }
  }
```

This code shows that the attributes are defined in AD objects as final static. The example groups a number of attributes together in an OCD object.

As can be seen from this example, the resource issues for using AttributeDefinition, ObjectClassDefinition and MetaTypeProvider classes are minimized.

4.7 Limitations

The OSGi MetaType specification is intended to be used for simple applications. It does not, therefore, support recursive data types, mixed types in arrays/vectors, or nested arrays/vectors.

4.8 Related Standards

One of the primary goals of this specification is to make metatype information available at run-time with minimal overhead. Many related standards are applicable to metatypes; except for Java beans, however, all other metatype standards are based on document formats (e.g. XML). In the OSGi Service Platform, document format standards are deemed unsuitable due to the overhead required in the execution environment (they require a parser during run-time).

Another consideration is the applicability of these standards. Most of these standards were developed for management systems on platforms where resources are not necessarily a concern. In this case, a metatype standard is normally used to describe the data structures needed to control some other computer via a network. This other computer, however, does not require the metatype information as it is *implementing* this information.

In some traditional cases, a management system uses the metatype information to control objects in an OSGi Service Platform. Therefore, the concepts and the syntax of the metatype information must be mappable to these popular standards. Clearly, then, these standards must be able to describe objects in an OSGi Service Platform. This ability is usually not a problem, because the metatype languages used by current management systems are very powerful.

4.8.1 Beans

The intention of the Beans packages in Java comes very close to the metatype information needed in the OSGi Service Platform. The java.beans.- packages cannot be used, however, for the following reasons:

- Beans packages require a large number of classes that are likely to be optional for an OSGi Service Platform.

- Beans have been closely coupled to the graphic subsystem (AWT) and applets. Neither of these packages is available on an OSGi Service Platform.
- Beans are closely coupled with the type-safe Java classes. The advantage of attributes is that no type-safety is used, allowing two parties to have an independent versioning model (no shared classes).
- Beans packages allow all possible Java objects, not the OSGi subset as required by this specification.
- Beans have no explicit localization.
- Beans have no support for optional attributes.

4.9 Security Considerations

Special security issues are not applicable for this specification.

4.10 Changes

This specification has not been changed since the previous release.

4.11 org.osgi.service.metatype

The OSGi Metatype Package. Specification Version 1.1.

Bundles wishing to use this package must list the package in the `Import-Package` header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.metatype; specification-ver-
sion=1.1
```

4.11.1 Summary

- AttributeDefinition - An interface to describe an attribute. [p.72]
- MetaTypeInformation - A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle. [p.75]
- MetaTypeProvider - Provides access to metatypes. [p.76]
- MetaTypeService - The MetaType Service can be used to obtain meta type information for a bundle. [p.76]
- ObjectClassDefinition - Description for the data type information of an objectclass. [p.67]

4.11.2 public interface AttributeDefinition

An interface to describe an attribute.

An `AttributeDefinition` object defines a description of the data type of a property/attribute.

4.11.2.1 public static final int BIGDECIMAL = 10

The `BIGDECIMAL` (10) type. Attributes of this type should be stored as `BigDecimal`, `Vector` with `BigDecimal` or `BigDecimal[]` objects depending on `getCardinality()`.

Deprecated Since 1.1

4.11.2.2 public static final int BIGINTEGER = 9

The `BIGINTEGER` (9) type. Attributes of this type should be stored as `BigInteger`, `Vector` with `BigInteger` or `BigInteger[]` objects, depending on the `getCardinality()` value.

Deprecated Since 1.1

4.11.2.3 public static final int BOOLEAN = 11

The `BOOLEAN` (11) type. Attributes of this type should be stored as `Boolean`, `Vector` with `Boolean` or `boolean[]` objects depending on `getCardinality()`.

4.11.2.4 public static final int BYTE = 6

The `BYTE` (6) type. Attributes of this type should be stored as `Byte`, `Vector` with `Byte` or `byte[]` objects, depending on the `getCardinality()` value.

4.11.2.5 public static final int CHARACTER = 5

The `CHARACTER` (5) type. Attributes of this type should be stored as `Character`, `Vector` with `Character` or `char[]` objects, depending on the `getCardinality()` value.

4.11.2.6 public static final int DOUBLE = 7

The `DOUBLE` (7) type. Attributes of this type should be stored as `Double`, `Vector` with `Double` or `double[]` objects, depending on the `getCardinality()` value.

4.11.2.7 public static final int FLOAT = 8

The `FLOAT` (8) type. Attributes of this type should be stored as `Float`, `Vector` with `Float` or `float[]` objects, depending on the `getCardinality()` value.

4.11.2.8 public static final int INTEGER = 3

The `INTEGER` (3) type. Attributes of this type should be stored as `Integer`, `Vector` with `Integer` or `int[]` objects, depending on the `getCardinality()` value.

4.11.2.9 public static final int LONG = 2

The `LONG` (2) type. Attributes of this type should be stored as `Long`, `Vector` with `Long` or `long[]` objects, depending on the `getCardinality()` value.

4.11.2.10 public static final int SHORT = 4

The `SHORT` (4) type. Attributes of this type should be stored as `Short`, `Vector` with `Short` or `short[]` objects, depending on the `getCardinality()` value.

4.11.2.11 public static final int STRING = 1

The `STRING` (1) type.

Attributes of this type should be stored as `String`, `Vector` with `String` or `String[]` objects, depending on the `getCardinality()` value.

4.11.2.12 public int getCardinality( )

□ Return the cardinality of this attribute. The OSGi environment handles multi valued attributes in arrays ([]) or in `Vector` objects. The return value is defined as follows:

```
    x = Integer.MIN_VALUE    no limit, but use Vector
    x < 0                    -x = max occurrences, store in 
Vector
    x > 0                     x = max occurrences, store in 
array []
    x = Integer.MAX_VALUE    no limit, but use array []
    x = 0                     1 occurrence required
```

4.11.2.13 public String[] getDefaultValue( )

□ Return a default for this attribute. The object must be of the appropriate type as defined by the cardinality and `getType()`. The return type is a list of `String` objects that can be converted to the appropriate type. The cardinality of the return array must follow the absolute cardinality of this type. E.g. if the cardinality = 0, the array must contain 1 element. If the cardinality is 1, it must contain 0 or 1 elements. If it is -5, it must contain from 0 to max 5 elements. Note that the special case of a 0 cardinality, meaning a single value, does not allow arrays or vectors of 0 elements.

Returns Return a default value or `null` if no default exists.

4.11.2.14 public String getDescription( )

□ Return a description of this attribute. The description may be localized and must describe the semantics of this type and any constraints.

Returns The localized description of the definition.

4.11.2.15 public String getID( )

□ Unique identity for this attribute. Attributes share a global namespace in the registry. E.g. an attribute `cn` or `commonName` must always be a `String` and the semantics are always a name of some object. They share this aspect with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify an attribute. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a Java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined attributes already have an OID. It is strongly advised to define the attributes from existing LDAP schemes which will give the OID. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns The id or oid

4.11.2.16 public String getName( )

□ Get the name of the attribute. This name may be localized.

Returns The localized name of the definition.

4.11.2.17 public String[] getOptionLabels( )

□ Return a list of labels of option values.

The purpose of this method is to allow menus with localized labels. It is associated with `getOptionValues`. The labels returned here are ordered in the same way as the values in that method.

If the function returns `null`, there are no option labels available.

This list must be in the same sequence as the `getOptionValues()` method. I.e. for each index i in `getOptionLabels`, i in `getOptionValues()` should be the associated value.

For example, if an attribute can have the value male, female, unknown, this list can return (for dutch) `new String[] { “Man”, “Vrouw”, “Onbekend” }`.

Returns A list values

4.11.2.18 public String[] getOptionValues( )

□ Return a list of option values that this attribute can take.

If the function returns `null`, there are no option values available.

Each value must be acceptable to validate() (return “”) and must be a `String` object that can be converted to the data type defined by getType() for this attribute.

This list must be in the same sequence as `getOptionLabels()`. I.e. for each index i in `getOptionValues`, i in `getOptionLabels()` should be the label.

For example, if an attribute can have the value male, female, unknown, this list can return `new String[] { “male”, “female”, “unknown” }`.

Returns A list values

4.11.2.19 public int getType( )

□ Return the type for this attribute.

Defined in the following constants which map to the appropriate Java type. `STRING`,`LONG`,`INTEGER`, `CHAR`,`BYTE`,`DOUBLE`,`FLOAT`, `BOOLEAN`.

4.11.2.20 public String validate(String value)

value The value before turning it into the basic data type

□ Validate an attribute in `String` form. An attribute might be further constrained in value. This method will attempt to validate the attribute according to these constraints. It can return three different values:

```
  null
                   no validation present
  “”                   no problems detected
  “...”                A localized description of why the
value is wrong
```

Returns `null`, “”, or another string

4.11.3 public interface MetaTypeInformation extends MetaTypeProvider

A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle.

4.11.3.1 public Bundle getBundle( )

□ Return the bundle for which this object provides metatype information.

Returns Bundle for which this object provides metatype information.

4.11.3.2 public String[] getFactoryPids( )

□ Return the Factory PIDs (for ManagedServices) for which ObjectClassDefinition information is available.

Returns Array of Factory PIDs.

4.11.3.3 public String[] getPids( )

□ Return the PIDs (for ManagedServices) for which ObjectClassDefinition information is available.

Returns Array of PIDs.

4.11.4 public interface MetaTypeProvider

Provides access to metatypes.

4.11.4.1 public String[] getLocales( )

□ Return a list of available locales. The results must be names that consists of language [_ country [_ variation]] as is customary in the `Locale` class.

Returns An array of locale strings or `null` if there is no locale specific localization can be found.

4.11.4.2 public ObjectClassDefinition getObjectClassDefinition(String id, String locale)

id The ID of the requested object class. This can be a pid or factory pid returned by getPids or getFactoryPids.

locale The locale of the definition or `null` for default locale.

□ Returns an object class definition for the specified id localized to the specified locale.

The locale parameter must be a name that consists of `language`["_" `country`["_" `variation`]] as is customary in the `Locale` class. This `Locale` class is not used because certain profiles do not contain it.

Returns A `ObjectClassDefinition` object.

Throws `IllegalArgumentException` – If the id or locale arguments are not valid

4.11.5 public interface MetaTypeService

The MetaType Service can be used to obtain meta type information for a bundle. The MetaType Service will examine the specified bundle for meta type documents and to create the returned MetaTypeInformation object.

4.11.5.1 public MetaTypeInformation getMetaTypeInformation(Bundle bundle)

bundle The bundle for which meta type information is requested.

□ Return the MetaType information for the specified bundle.

Returns MetaTypeInformation object for the specified bundle.

4.11.6 public interface ObjectClassDefinition

Description for the data type information of an objectclass.

4.11.6.1 public static final int ALL = -1

Argument for `getAttributeDefinitions(int)`.

`ALL` indicates that all the definitions are returned. The value is -1.

4.11.6.2 public static final int OPTIONAL = 2

Argument for `getAttributeDefinitions(int)`.

`OPTIONAL` indicates that only the optional definitions are returned. The value is 2.

4.11.6.3 public static final int REQUIRED = 1

Argument for `getAttributeDefinitions(int)`.

`REQUIRED` indicates that only the required definitions are returned. The value is 1.

4.11.6.4 public AttributeDefinition[] getAttributeDefinitions(int filter)

filter `ALL`,`REQUIRED`,`OPTIONAL`

□ Return the attribute definitions for this object class.

Return a set of attributes. The filter parameter can distinguish between `ALL`, `REQUIRED` or the `OPTIONAL` attributes.

Returns An array of attribute definitions or `null` if no attributes are selected

4.11.6.5 public String getDescription( )

□ Return a description of this object class. The description may be localized.

Returns The description of this object class.

4.11.6.6 public InputStream getIcon(int size) throws IOException

size Requested size of an icon, e.g. a 16x16 pixels icon then size = 16

□ Return an `InputStream` object that can be used to create an icon from.

Indicate the size and return an `InputStream` object containing an icon. The returned icon maybe larger or smaller than the indicated size.

The icon may depend on the localization.

Returns An InputStream representing an icon or `null`

4.11.6.7 public String getID( )

□ Return the id of this object class.

`ObjectDefintion` objects share a global namespace in the registry. They share this aspect with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined object classes already have an OID associated. It is strongly advised to define the object classes from existing LDAP schemes which will give the OID for free. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns The id of this object class.

4.11.6.8 public String getName( )

- Return the name of this object class. The name may be localized.

Returns The name of this object class.

4.12 References

[10] *LDAP.*
Available at http://directory.google.com/Top/Computers/Software/Internet/Servers/Directory/LDAP

[11] *Understanding and Deploying LDAP Directory services*
Timothy Howes et. al. ISBN 1-57870-070-1, MacMillan Technical publishing.

7 Event Service Specification

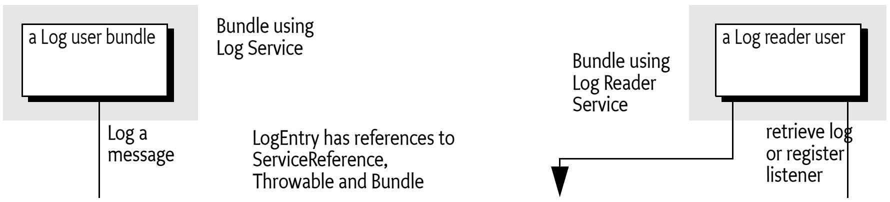
Version 1.0

7.1 Introduction

7.1.1 Entities

- *Application* – ....
- *Application Descrriptor* –

Figure 19 Log Service Class Diagram org.osgi.service.log package


a Log user bundle
Bundle using Log Service
Log a message
LogEntry has references to ServiceReference, Throwable and Bundle
Bundle using Log Reader Service
a Log reader user
retrieve log or register listener

7.2 The Event Service

7.3 Security

7.4 org.osgi.service.event

The OSGi Event Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.event; specification-ver-
sion=1.0
```

7.4.1 Summary

- ChannelEvent - [p.95]
- ChannelListener - [p.96]
- EventChannel - [p.96]

7.4.2 public class ChannelEvent

7.4.2.1 public ChannelEvent(String topic, Map properties)

7.4.2.2 public boolean equals(Object obj)

7.4.2.3 public final Object getProperty(String name)

7.4.2.4 public final String[] getPropertyNames( )

7.4.2.5 public final String getTopic( )

7.4.2.6 public int hashCode( )

7.4.2.7 public final boolean matches(Filter filter)

7.4.2.8 public String toString( )

7.4.3 public interface ChannelListener extends EventListener

7.4.3.1 public void channelEvent(ChannelEvent event)

7.4.4 public interface EventChannel

7.4.4.1 public void postEvent(ChannelEvent event)

7.4.4.2 public void sendEvent(ChannelEvent event)

6 IO Connector Service Specification

Version 1.0

6.1 Introduction

Communication is at the heart of OSGi Service Platform functionality. Therefore, a flexible and extendable communication API is needed: one that can handle all the complications that arise out of the Reference Architecture. These obstacles could include different communication protocols based on different networks, firewalls, intermittent connectivity, and others.

Therefore, this IO Connector Service specification adopts the [12] *Java 2 Micro Edition* (J2ME) javax.microedition.io packages as a basic communications infrastructure. In J2ME, this API is also called the Connector framework. A key aspect of this framework is that the connection is configured by a single string, the URI.

In J2ME, the Connector framework can be extended by the vendor of the Virtual Machine, but cannot be extended at run-time by other code. Therefore, this specification defines a service that adopts the flexible model of the Connector framework, but allows bundles to extend the Connector Services into different communication domains.

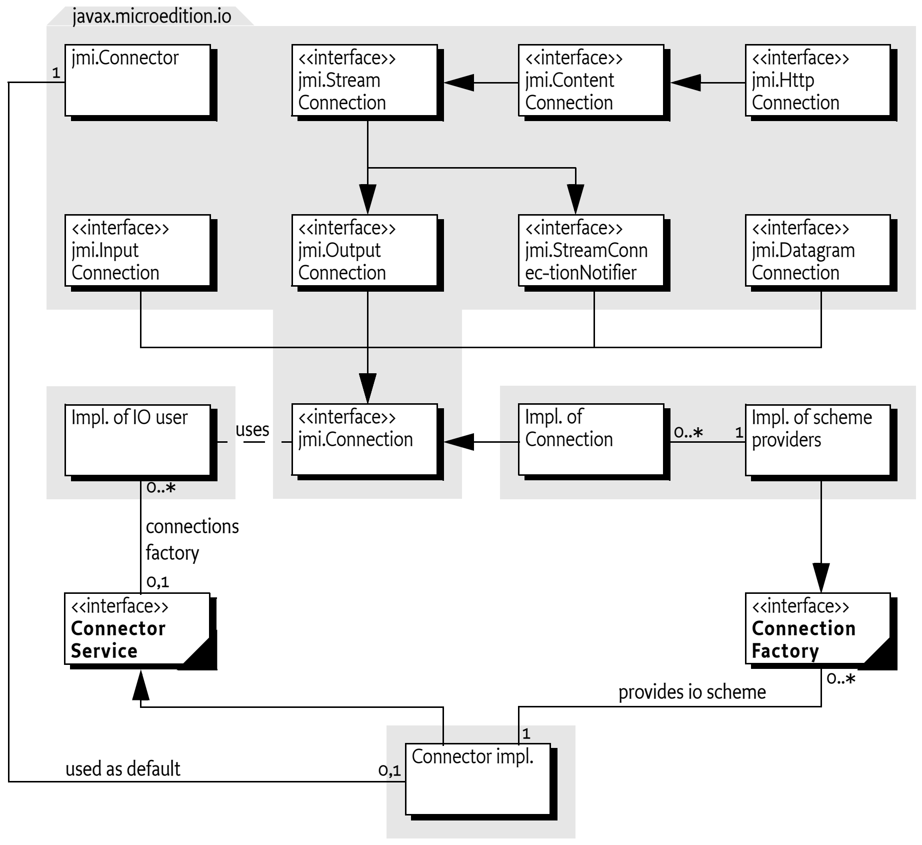
6.1.1 Essentials

- *Abstract* – Provide an intermediate layer that abstracts the actual protocol and devices from the bundle using it.
- *Extendable* – Allow third-party bundles to extend the system with new protocols and devices.
- *Layered* – Allow a protocol to be layered on top of lower layer protocols or devices.
- *Configurable* – Allow the selection of an actual protocol/device by means of configuration data.
- *Compatibility* – Be compatible with existing standards.

6.1.2 Entities

- *ConnectorService* – The service that performs the same function—creating connections from different providers—as the static methods in the Connector framework of javax.microediton.io.
- *ConnectionFactory* – A service that extends the Connector service with more schemes.
- *Scheme* – A protocol or device that is supported in the Connector framework.

Figure 18 *Class Diagram, org.osgi.service.io (jmi is javax.microedition.io)*


javax.microedition.io
jmi.Connector
1
<<interface>> jmi.Stream Connection
<<interface>> jmi.Content Connection
<<interface>> jmi.Http Connection
<<interface>> jmi.Input Connection
<<interface>> jmi.Output Connection
<<interface>> jmi.StreamConn ec-tionNotifier
<<interface>> jmi.Datagram Connection
Impl. of IO user
uses
<<interface>> jmi.Connection
Impl. of Connection
0..*
1
Impl. of scheme providers
0..*
connections factory
0,1
<<interface>> Connector Service
<<interface>> Connection Factory
0..*
provides io scheme
1
used as default
0,1
Connector impl.

6.2 The Connector Framework

The [12] *Java 2 Micro Edition* specification introduces a package for communicating with back-end systems. The requirements for this package are very similar to the following OSGi requirements:

- Small footprint
- Allows many different implementations simultaneously
- Simple to use
- Simple configuration

The key design goal of the Connector framework is to allow an application to use a communication mechanism/protocol without understanding implementation details.

An application passes a Uniform Resource Identifier (URI) to the java.microedition.io.Connector class, and receives an object implementing one or more Connection interfaces. The java.microedition.io.Connector class uses the scheme in the URI to locate the appropriate Connection Factory service. The remainder of the URI may contain parameters that are used by the Connection Factory service to establish the connection; for example, they may contain the baud rate for a serial connection. Some examples:

- sms://+46705950899;expiry=24h;reply=yes;type=9
- datagram://:53
- socket://www.acme.com:5302
- comm://COM1;baudrate=9600;databits=9
- file:c:/autoexec.bat

The javax.microedition.io API itself does not prescribe any schemes. It is up to the implementor of this package to include a number of extensions that provide the schemes. The javax.microedition.io.Connector class dispatches a request to a class which provides an implementation of a Connection interface. J2ME does not specify how this dispatching takes place, but implementations usually offer a proprietary mechanism to connect user defined classes that can provide new schemes.

The Connector framework defines a taxonomy of communication mechanisms with a number of interfaces. For example, a javax.microedition.io.InputConnection interface indicates that the connection supports the input stream semantics, such as an I/O port. A javax.microedition.io.DatagramConnection interface indicates that communication should take place with messages.

When a javax.microedition.io.Connector.open method is called, it returns a javax.microedition.io.Connection object. The interfaces implemented by this object define the type of the communication session. The following interfaces may be implemented:

- *HttpConnection* – A javax.microedition.io.ContentConnection with specific HTTP support.
- *DatagramConnection* – A connection that can be used to send and receive datagrams.
- *OutputConnection* – A connection that can be used for streaming output.
- *InputConnection* – A connection that can be used for streaming input.
- *StreamConnection* – A connection that is both input and output.
- *StreamConnectionNotifier* – Can be used to wait for incoming stream connection requests.
- *ContentConnection* – A javax.microedition.io.StreamConnection that provides information about the type, encoding, and length of the information.

Bundles using this approach must indicate to the Operator what kind of interfaces they expect to receive. The operator must then configure the bundle with a URI that contains the scheme and appropriate options that match the bundle's expectations. Well-written bundles are flexible enough to communicate with any of the types of javax.microedition.io.Connection interfaces they have specified. For example, a bundle should support javax.microedition.io.StreamConnection as well as javax.microedition.io.DatagramConnection objects in the appropriate direction (input or output).

The following code example shows a bundle that sends an alarm message with the help of the javax.microedition.io.Connector framework:

```
public class Alarm {
  String    uri;
  public Alarm(String uri) { this.uri = uri; }
  private void send(byte[] msg) {
        while ( true ) try {
           Connection  connection = Connector.open( uri );
           DataOutputStream    dout = null;
           if ( connection instanceof OutputConnection ) {
              dout = ((OutputConnection)
                 connection).openDataOutputStream();
             dout.write( msg );
           }
           else if (connection instanceof DatagramConnection) {
             DatagramConnection dgc =
                 (DatagramConnection) connection;
             Datagram datagram = dgc.newDatagram(
                 msg, msg.length );
             dgc.send( datagram );
           } else {
               error( "No configuration for alarm" );
               return;
           }
           connection.close();
         } catch( Exception e ) { ... }
      }
   }
```

6.3 Connector Service

The javax.microedition.io.Connector framework matches the requirements for OSGi applications very well. The actual creation of connections, however, is handled through static methods in the javax.microedition.io.Connector class. This approach does not mesh well with the OSGi service registry and dynamic life-cycle management.

This specification therefore introduces the Connector Service. The methods of the ConnectorService interface have the same signatures as the static methods of the javax.microedition.io.Connector class.

Each javax.microedition.io.Connection object returned by a Connector Service must implement interfaces from the javax.microedition.io package. Implementations must strictly follow the semantics that are associated with these interfaces.

The Connector Service must provide all the schemes provided by the exporter of the javax.microedition.io package. The Connection Factory services must have priority over schemes implemented in the Java run-time environment. For example, if a Connection Factory provides the http scheme and a built-in implementation exists, then the Connector Service must use the Connection Factory service with the http scheme.

Bundles that want to use the Connector Service should first obtain a ConnectorService service object. This object contains open methods that should be called to get a new javax.microedition.io.Connection object.

6.4 Providing New Schemes

The Connector Service must be able to be extended with the Connection Factory service. Bundles that can provide new schemes must register a ConnectionFactory service object.

The Connector Service must listen for registrations of new ConnectionFactory service objects and make the supplied schemes available to bundles that create connections.

Implementing a Connection Factory service requires implementing the following method:

- createConnection(String,int,boolean) – Creates a new connection object from the given URI.

The Connection Factory service must be registered with the IO_SCHEME property to indicate the provided scheme to the Connector Service. The value of this property must be a String[] object.

If multiple Connection Factory services register with the same scheme, the Connector Service should select the Connection Factory service with the highest value for the service.ranking service registration property, or if more than one Connection Factory service has the highest value, the Connection Factory service with the lowest service.id is selected.

The following example shows how a Connection Factory service may be implemented. The example will return a javax.microedition.io.InputConnection object that returns the value of the URI after removing the scheme identifier.

```
public class ConnectionFactoryImpl
   implements BundleActivator, ConnectionFactory {
      public void start( BundleContext context ) {
         Hashtable  properties = new Hashtable();
         properties.put( IO_SCHEME,
            new String[] { "data" } );
         context.registerService(
            ConnectorService.class.getName(),
            this, properties );
      }
      public void stop( BundleContext context ) {}

      public Connection createConnection(
         String uri, int mode, boolean timeouts  ) {
         return new DataConnection(uri);
      }
}

class DataConnection
   implements javax.microedition.io.InputConnection {
   String    uri;
   DataConnection( String uri ) {this.uri = uri;}
   public DataInputStream openDataInputStream()
      throws IOException {
```

```
      return new DataInputStream( openInputStream() );
   }

   public InputStream openInputStream() throws IOException {
      byte [] buf = uri.getBytes();
      return new ByteArrayInputStream(buf,5,buf.length-5);
   }
   public void close() {}
}
```

6.4.1 Orphaned Connection Objects

When a Connection Factory service is unregistered, it must close all Connection objects that are still open. Closing these Connection objects should make these objects unusable, and they should subsequently throw an IOException when used.

Bundles should not unnecessarily hang onto objects they retrieved from services. Implementations of Connection Factory services should program defensively and ensure that resource allocation is minimized when a Connection object is closed.

6.5 Execution Environment

The javax.microedition.io package is available in J2ME configurations/profiles, but is not present in J2SE, J2EE, and the OSGi minimum execution requirements.

Implementations of the Connector Service that are targeted for all environments should carry their own implementation of the javax.microedition.io package and export it.

6.6 Security

The OSGi Connector Service is a key service available in the Service Platform. A malicious bundle which provides this service can spoof any communication. Therefore, it is paramount that the ServicePermission[REGISTER,ConnectorService] is given only to a trusted bundle. ServicePermission[GET,ConnectorService] may be handed to bundles that are allowed to communicate to the external world.

ServicePermission[REGISTER,ConnectionFactory] should also be restricted to trusted bundles because they can implement specific protocols or access devices. ServicePermission[GET,ConnectionFactory] should be limited to trusted bundles that implement the Connector Service.

Implementations of Connection Factory services must perform all I/O operations within a privileged region. For example, an implementation of the sms: scheme must have permission to access the mobile phone, and should not require the bundle that opened the connection to have this permission. Normally, the operations need to be implemented in a doPrivileged method or in a separate thread.

If a specific Connection Factory service needs more detailed permissions than provided by the OSGi or Java 2, it may create a new specific Permission sub-class for its purpose.

6.7 org.osgi.service.io

The OSGi IO Connector Specification Version 1.0.

Bundles wishing to use this package must list the package in the `Import-Package` header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.io; specification-ver-
sion=1.0, javax.microedition.io
```

6.7.1 Summary

- ConnectionFactory - A Connection Factory service is called by the implementation of the Connector Service to create `javax.microedition.io.Connection` objects which implement the scheme named by `IO_SCHEME`. [p.89]
- ConnectorService - The Connector Service should be called to create and open `javax.microedition.io.Connection` objects. [p.91]

6.7.2 public interface ConnectionFactory

A Connection Factory service is called by the implementation of the Connector Service to create `javax.microedition.io.Connection` objects which implement the scheme named by `IO_SCHEME`. When a `ConnectorService.open` method is called, the implementation of the Connector Service will examine the specified name for a scheme. The Connector Service will then look for a Connection Factory service which is registered with the service property `IO_SCHEME` which matches the scheme. The `createConnection`[p.91] method of the selected Connection Factory will then be called to create the actual `Connection` object.

6.7.2.1 public static final String IO_SCHEME = "io.scheme"

Service property containing the scheme(s) for which this Connection Factory can create `Connection` objects. This property is of type `String[]`.

6.7.2.2 public Connection createConnection(String name, int mode, boolean timeouts) throws IOException

name The full URI passed to the `ConnectorService.open` method

mode The mode parameter passed to the `ConnectorService.open` method

timeouts The timeouts parameter passed to the `ConnectorService.open` method

□ Create a new `Connection` object for the specified URI.

Returns A new `javax.microedition.io.Connection` object.

Throws `IOException` – If a `javax.microedition.io.Connection` object can not not be created.

6.7.3 public interface ConnectorService

The Connector Service should be called to create and open `javax.microedition.io.Connection` objects. When an `open*` method is called, the implementation of the Connector Service will examine the specified name for a scheme. The Connector Service will then look for a Connection Factory service which is registered with the service property `IO_SCHEME` which matches the scheme. The `createConnection` method of the selected Connection Factory will then be called to create the actual `Connection` object.

If more than one Connection Factory service is registered for a particular scheme, the service with the highest ranking (as specified in its `service.ranking` property) is called. If there is a tie in ranking, the service with the lowest service ID (as specified in its `service.id` property), that is the service that was registered first, is called. This is the same algorithm used by `BundleContext.getServiceReference`.

6.7.3.1 public static final int READ = 1

Read access mode.

See Also `javax.microedition.io.Connector.READ`

6.7.3.2 public static final int READ_WRITE = 3

Read/Write access mode.

See Also `javax.microedition.io.Connector.READ_WRITE`

6.7.3.3 public static final int WRITE = 2

Write access mode.

See Also `javax.microedition.io.Connector.WRITE`

6.7.3.4 public Connection open(String name) throws IOException

name The URI for the connection.

□ Create and open a `Connection` object for the specified name.

Returns A new `javax.microedition.io.Connection` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.open(String name)`

6.7.3.5 public Connection open(String name, int mode) throws IOException

name The URI for the connection.

mode The access mode.

□ Create and open a `Connection` object for the specified name and access mode.

Returns A new `javax.microedition.io.Connection` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.open(String name, int mode)`

6.7.3.6 public Connection open(String name, int mode, boolean timeouts) throws IOException

name The URI for the connection.

mode The access mode.

timeouts A flag to indicate that the caller wants timeout exceptions.

□ Create and open a `Connection` object for the specified name, access mode and timeouts.

Returns A new `javax.microedition.io.Connection` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.open(String name, int mode, boolean timeouts)`

6.7.3.7 public DataInputStream openDataInputStream(String name) throws IOException

name The URI for the connection.

□ Create and open a `DataInputStream` object for the specified name.

Returns A `DataInputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openDataInputStream(String name)`

6.7.3.8 public DataOutputStream openDataOutputStream(String name) throws IOException

name The URI for the connection.

□ Create and open a `DataOutputStream` object for the specified name.

Returns A `DataOutputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openDataOutputStream(String name)`

6.7.3.9 public InputStream openInputStream(String name) throws IOException

name The URI for the connection.

□ Create and open an `InputStream` object for the specified name.

Returns An `InputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openInputStream(String name)`

6.7.3.10 public OutputStream openOutputStream(String name) throws IOException

name The URI for the connection.

□ Create and open an `OutputStream` object for the specified name.

Returns An `OutputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openOutputStream(String name)`

6.8 References

[12] *Java 2 Micro Edition*
http://java.sun.com/j2me/

[13] *javax.microedition.io whitepaper*
http://wireless.java.sun.com/midp/chapters/j2mewhite/chap13.pdf

[14] *J2ME Foundation Profile*
http://www.jcp.org/jsr/detail/46.jsp

8 Device Access Specification

Version 1.1

8.1 Introduction

A Service Platform is a meeting point for services and devices from many different vendors: a meeting point where users add and cancel service subscriptions, newly installed services find their corresponding input and output devices, and device drivers connect to their hardware.

In an OSGi Service Platform, these activities will dynamically take place while the Framework is running. Technologies such as USB and IEEE 1394 explicitly support plugging and unplugging devices at any time, and wireless technologies are even more dynamic.

This flexibility makes it hard to configure all aspects of an OSGi Service Platform, particularly those relating to devices. When all of the possible services and device requirements are factored in, each OSGi Service Platform will be unique. Therefore, automated mechanisms are needed that can be extended and customized, in order to minimize the configuration needs of the OSGi environment.

The Device Access specification supports the coordination of automatic detection and attachment of existing devices on an OSGi Service Platform, facilitates hot-plugging and -unplugging of new devices, and downloads and installs device drivers on demand.

This specification, however, deliberately does not prescribe any particular device or network technology, and mentioned technologies are used as examples only. Nor does it specify a particular device discovery method. Rather, this specification focuses on the attachment of devices supplied by different vendors. It emphasizes the development of standardized device interfaces to be defined in device categories, although no such device categories are defined in this specification.

8.1.1 Essentials

- *Embedded Devices* – OSGi bundles will likely run in embedded devices. This environment implies limited possibility for user interaction, and low-end devices will probably have resource limitations.
- *Remote Administration* – OSGi environments must support administration by a remote service provider.
- *Vendor Neutrality* – OSGi-compliant driver bundles will be supplied by different vendors; each driver bundle must be well-defined, documented, and replaceable.

- *Continuous Operation* – OSGi environments will be running for extended periods without being restarted, possibly continuously, requiring stable operation and stable resource consumption.
- *Dynamic Updates* – As much as possible, driver bundles must be individually replaceable without affecting unrelated bundles. In particular, the process of updating a bundle should not require a restart of the whole OSGi Service Platform or disrupt operation of connected devices.

A number of requirements must be satisfied by Device Access implementations in order for them to be OSGi-compliant. Implementations must support the following capabilities:

- *Hot-Plugging* – Plugging and unplugging of devices at any time if the underlying hardware and drivers allow it.
- *Legacy Systems* – Device technologies which do not implement the automatic detection of plugged and unplugged devices.
- *Dynamic Device Driver Loading* – Loading new driver bundles on demand with no prior device-specific knowledge of the Device service.
- *Multiple Device Representations* – Devices to be accessed from multiple levels of abstraction.
- *Deep Trees* – Connections of devices in a tree of mixed network technologies of arbitrary depth.
- *Topology Independence* – Separation of the interfaces of a device from where and how it is attached.
- *Complex Devices* – Multifunction devices and devices that have multiple configurations.

8.1.2 Operation

This specification defines the behavior of a device manager (which is *not* a service as might be expected). This device manager detects registration of Device services and is responsible for associating these devices with an appropriate Driver service. These tasks are done with the help of Driver Locator services and the Driver Selector service that allow a device manager to find a Driver bundle and install it.

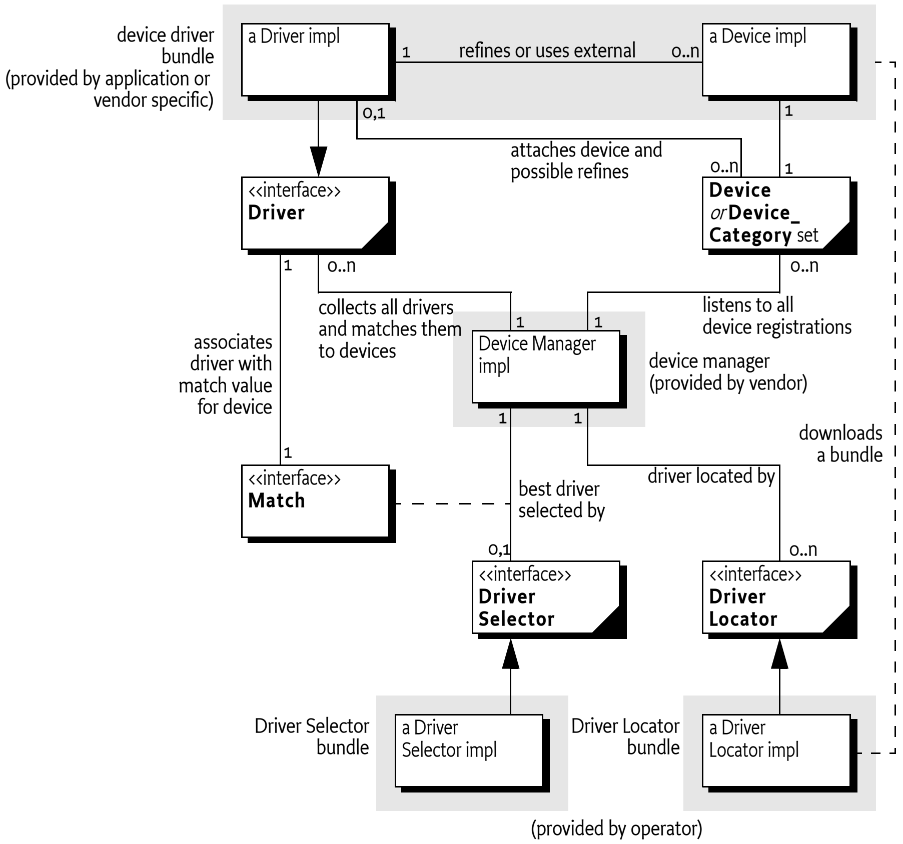
8.1.3 Entities

The main entities of the Device Access specification are:

- *Device Manager* – The bundle that controls the initiation of the attachment process behind the scenes.
- *Device Category* – Defines how a Driver service and a Device service can cooperate.
- *Driver* – Competes for attaching Device services of its recognized device category. See *Driver Services* on page 104.
- *Device* – A representation of a physical device or other entity that can be attached by a Driver service. See *Device Services* on page 99.
- *DriverLocator* – Assists in locating bundles that provide a Driver service. See *Driver Locator Service* on page 111.
- *DriverSelector* – Assists in selecting which Driver service is best suited to a Device service. See *The Driver Selector Service* on page 113.

Figure 20 show the classes and their relationships.

Figure 20 *Device Access Class Overview*


device driver bundle (provided by application or vendor specific)
a Driver impl
refines or uses external
1
0..n
a Device impl
0,1
1
attaches device and possible refines
0..n
1
<<interface>> Driver
Device or Device_ Category set
1
0..n
0..n
collects all drivers and matches them to devices
listens to all device registrations
associates driver with match value for device
1
1
Device Manager impl
device manager (provided by vendor)
1
1
downloads a bundle
1
<<interface>> Match
best driver selected by
driver located by
0,1
0..n
<<interface>> Driver Selector
<<interface>> Driver Locator
Driver Selector bundle
a Driver Selector impl
Driver Locator bundle
a Driver Locator impl
(provided by operator)

8.2 Device Services

A Device service represents some form of a device. It can represent a hardware device, but that is not a requirement. Device services differ widely: some represent individual physical devices and others represent complete networks. Several Device services can even simultaneously represent the same physical device at different levels of abstraction. For example:

- A USB network.
- A device attached on the USB network.
- The same device recognized as a USB to Ethernet bridge.
- A device discovered on the Ethernet using Salutation.
- The same device recognized as a simple printer.
- The same printer refined to a PostScript printer.

A device can also be represented in different ways. For example, a USB mouse can be considered as:

- A USB device which delivers information over the USB bus.
- A mouse device which delivers x and y coordinates and information about the state of its buttons.

Each representation has specific implications:

- That a particular device is a mouse is irrelevant to an application which provides management of USB devices.
- That a mouse is attached to a USB bus or a serial port would be inconsequential to applications that respond to mouse-like input.

Device services must belong to a defined *device category*, or else they can implement a generic service which models a particular device, independent of its underlying technology. Examples of this type of implementation could be Sensor or Actuator services.

A device category specifies the methods for communicating with a Device service, and enables interoperability between bundles that are based on the same underlying technology. Generic Device services will allow interoperability between bundles that are not coupled to specific device technologies.

For example, a device category is required for the USB, so that Driver bundles can be written that communicate to the devices that are attached to the USB. If a printer is attached, it should also be available as a generic Printer service defined in a Printer service specification, indistinguishable from a Printer service attached to a parallel port. Generic categories, such as a Printer service, should also be described in a Device Category.

It is expected that most Device service objects will actually represent a physical device in some form, but that is not a requirement of this specification. A Device service is represented as a normal service in the OSGi Framework and all coordination and activities are performed upon Framework services. This specification does not limit a bundle developer from using Framework mechanisms for services that are not related to physical devices.

8.2.1 Device Service Registration

A Device service is defined as a normal service registered with the Framework that either:

- Registers a service object under the interface org.osgi.service.Device with the Framework, or
- Sets the DEVICE_CATEGORY property in the registration. The value of DEVICE_CATEGORY is an array of String objects of all the device categories that the device belongs to. These strings are defined in the associated device category.

If this document mentions a Device service, it is meant to refer to services registered with the name org.osgi.service.device.Device *or* services registered with the DEVICE_CATEGORY property set.

When a Device service is registered, additional properties may be set that describe the device to the device manager and potentially to the end users. The following properties have their semantics defined in this specification:

- DEVICE_CATEGORY – A marker property indicating that this service must be regarded as a Device service by the device manager. Its value is of type String[], and its meaning is defined in the associated device category specification.
- DEVICE_DESCRIPTION – Describes the device to an end user. Its value is of type String.

- DEVICE_SERIAL – A unique serial number for this device. If the device hardware contains a serial number, the driver bundle is encouraged to specify it as this property. Different Device services representing the same physical hardware at different abstraction levels should set the same DEVICE_SERIAL, thus simplifying identification. Its value is of type String.
- service.pid – Service Persistent ID (PID), defined in org.osgi.framework.Constants. Device services should set this property. It must be unique among all registered services. Even different abstraction levels of the same device must use different PIDs. The service PIDs must be reproducible, so that every time the same hardware is plugged in, the same PIDs are used.

8.2.2 Device Service Attachment

When a Device service is registered with the Framework, the device manager is responsible for finding a suitable Driver service and instructing it to attach to the newly registered Device service. The Device service itself is passive: it only registers a Device service with the Framework and then waits until it is called.

The actual communication with the underlying physical device is not defined in the Device interface because it differs significantly between different types of devices. The Driver service is responsible for attaching the device in a device type-specific manner. The rules and interfaces for this process must be defined in the appropriate device category.

If the device manager is unable to find a suitable Driver service, the Device service remains unattached. In that case, if the service object implements the Device interface, it must receive a call to the noDriverFound() method. The Device service can wait until a new driver is installed, or it can unregister and attempt to register again with different properties that describe a more generic device or try a different configuration.

8.2.2.1 Idle Device Service

The main purpose of the device manager is to try to attach drivers to idle devices. For this purpose, a Device service is considered *idle* if no bundle that itself has registered a Driver service is using the Device service.

8.2.2.2 Device Service Unregistration

When a Device service is unregistered, no immediate action is required by the device manager. The normal service of unregistering events, provided by the Framework, takes care of propagating the unregistration information to affected drivers. Drivers must take the appropriate action to release this Device service and perform any necessary cleanup, as described in their device category specification.

The device manager may, however, take a device unregistration as an indication that driver bundles may have become idle and are thus eligible for removal. It is therefore important for Device services to unregister their service object when the underlying entity becomes unavailable.

8.3 Device Category Specifications

A device category specifies the rules and interfaces needed for the communication between a Device service and a Driver service. Only Device services and Driver services of the same device category can communicate and cooperate.

The Device Access service specification is limited to the attachment of Device services by Driver services, and does *not* enumerate different device categories.

Other specifications must specify a number of device categories before this specification can be made operational. Without a set of defined device categories, no interoperability can be achieved.

Device categories are related to a specific device technology, such as USB, IEEE 1394, JINI, UPnP, Salutation, CEBus, Lonworks, and others. The purpose of a device category specification is to make all Device services of that category conform to an agreed interface, so that, for example, a USB Driver service of vendor A can control Device services from vendor B attached to a USB bus.

This specification is limited to defining the guidelines for device category definitions only. Device categories may be defined by the OSGi organization or by external specification bodies – for example, when these bodies are associated with a specific device technology.

8.3.1 Device Category Guidelines

A device category definition comprises the following elements:

- An interface that all devices belonging to this category must implement. This interface should lay out the rules of how to communicate with the underlying device. The specification body may define its own device interfaces (or classes) or leverage existing ones. For example, a serial port device category could use the `javax.comm.SerialPort` interface which is defined in [15] *Java Communications API*.
  When registering a device belonging to this category with the Framework, the interface or class name for this category must be included in the registration.
- A set of service registration properties, their data types, and semantics, each of which must be declared as either `MANDATORY` or `OPTIONAL` for this device category.
- A range of match values specific to this device category. Matching is explained later in *The Device Attachment Algorithm* on page 115.

8.3.2 Sample Device Category Specification

The following is a partial example of a fictitious device category:

```
public interface /* com.acme.widget.*/ WidgetDevice {
   int MATCH_SERIAL                    = 10;
   int MATCH_VERSION                   =  8;
   int MATCH_MODEL                     =  6;
   int MATCH_MAKE                      =  4;
   int MATCH_CLASS                     =  2;
```

```
    void sendPacket( byte [] data );
    byte [] receivePacket( long timeout );
  }
```

Devices in this category must implement the interface com.acme.widget.WidgetDevice to receive attachments from Driver services in this category.

Device properties for this fictitious category are defined in table Table 9.

Property name	M/O	Type	Value
DEVICE_CATEGORY	M	String[]	{"Widget"}
com.acme.class	M	String	A class description of this device. For example "audio", "video", "serial", etc. An actual device category specification should contain an exhaustive list and define a process to add new classes.
com.acme.model	M	String	A definition of the model. This is usually vendor specific. For example "Mouse".
com.acme.manufacturer	M	String	Manufacturer of this device, for example "ACME Widget Division".
com.acme.revision	O	String	Revision number. For example, "42".
com.acme.serial	O	String	A serial number. For example "SN6751293-12-2112/A".

Table 9 Example Device Category Properties, M=Mandatory, O=Optional

8.3.3 Match Example

Driver services and Device services are connected via a matching process that is explained in *The Device Attachment Algorithm* on page 115. The Driver service plays a pivotal role in this matching process. It must inspect the Device service (from its ServiceReference object) that has just been registered and decide if it potentially could cooperate with this Device service.

It must be able to answer a value indicating the quality of the match. The scale of this match value must be defined in the device category so as to allow Driver services to match on a fair basis. The scale must start at least at 1 and go upwards.

Driver services for this sample device category must return one of the match codes defined in the com.acme.widget.WidgetDevice interface or Device.MATCH_NONE if the Device service is not recognized. The device category must define the exact rules for the match codes in the device category specification. In this example, a small range from 2 to 10 (MATCH_NONE is 0) is defined for WidgetDevice devices. They are named in the WidgetDevice interface for convenience and have the following semantics.

Match name	Value	Description
MATCH_SERIAL	10	An exact match, including the serial number.
MATCH_VERSION	8	Matches the right class, make model, and version.
MATCH_MODEL	6	Matches the right class and make model.
MATCH_MAKE	4	Matches the make.
MATCH_CLASS	2	Only matches the class.

Table 10 Sample Device Category Match Scale

A Driver service should use the constants to return when it decides how closely the Device service matches its suitability. For example, if it matches the exact serial number, it should return MATCH_SERIAL.

8.4 Driver Services

A Driver service is responsible for attaching to suitable Device services under control of the device manager. Before it can attach a Device service, however, it must compete with other Driver services for control.

If a Driver service wins the competition, it must attach the device in a device category-specific way. After that, it can perform its intended functionality. This functionality is not defined here nor in the device category; this specification only describes the behavior of the Device service, not how the Driver service uses it to implement its intended functionality. A Driver service may register one or more new Device services of another device category or a generic service which models a more refined form of the device.

Both refined Device services as well as generic services should be defined in a Device Category. See *Device Category Specifications* on page 102.

8.4.1 Driver Bundles

A Driver service is, like *all* services, implemented in a bundle, and is recognized by the device manager by registering one or more Driver service objects with the Framework.

Such bundles containing one or more Driver services are called *driver bundles*. The device manager must be aware of the fact that the cardinality of the relationship between bundles and Driver services is 1:1…*.

A driver bundle must register *at least* one Driver service in its BundleActivator.start implementation.

8.4.2 Driver Taxonomy

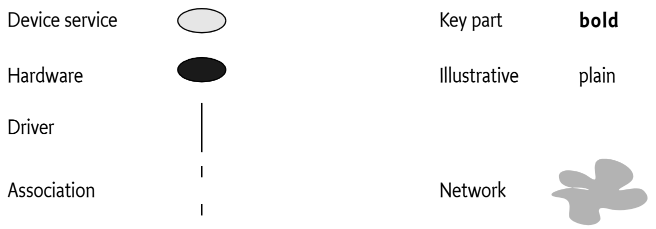
Device Drivers may belong to one of the following categories:

- Base Drivers (Discovery, Pure Discovery and Normal)
- Refining Drivers
- Network Drivers
- Composite Drivers

- Referring Drivers
- Bridging Drivers
- Multiplexing Drivers
- Pure Consuming Drivers

This list is not definitive, and a Driver service is not required to fit into one of these categories. The purpose of this taxonomy is to show the different topologies that have been considered for the Device Access service specification.

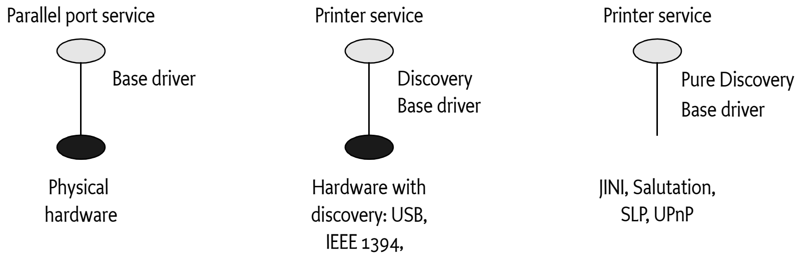
Figure 21 Legend for Device Driver Services Taxonomy



8.4.2.1 Base Drivers

The first category of device drivers are called *base drivers* because they provide the lowest-level representation of a physical device. The distinguishing factor is that they are not registered as Driver services because they do not have to compete for access to their underlying technology.

Figure 22 Base Driver Types



Base drivers discover physical devices using code not specified here (for example, through notifications from a device driver in native code) and then register corresponding Device services.

When the hardware supports a discovery mechanism and reports a physical device, a Device service is then registered. Drivers supporting a discovery mechanism are called *discovery base drivers*.

An example of a discovery base driver is a USB driver. Discovered USB devices are registered with the Framework as a generic USB Device service. The USB specification (see [16] *USB Specification*) defines a tightly integrated discovery method. Further, devices are individually addressed; no provision exists for broadcasting a message to all devices attached to the USB bus. Therefore, there is no reason to expose the USB network itself; instead, a discovery base driver can register the individual devices as they are discovered.

Not all technologies support a discovery mechanism. For example, most serial ports do not support detection, and it is often not even possible to detect whether a device is attached to a serial port.

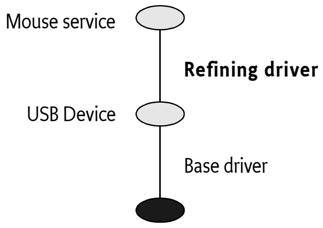
Although each driver bundle should perform discovery on its own, a driver for a non-discoverable serial port requires external help – either through a user interface or by allowing the Configuration Admin service to configure it.

It is possible for the driver bundle to combine automatic discovery of Plug and Play-compliant devices with manual configuration when non-compliant devices are plugged in.

8.4.2.2 Refining Drivers

The second category of device drivers are called *refining drivers*. Refining drivers provide a refined view of a physical device that is already represented by another Device service registered with the Framework. Refining drivers register a Driver service with the Framework. This Driver service is used by the device manager to attach the refining driver to a less refined Device service that is registered as a result of events within the Framework itself.

Figure 23 Refining Driver Diagram



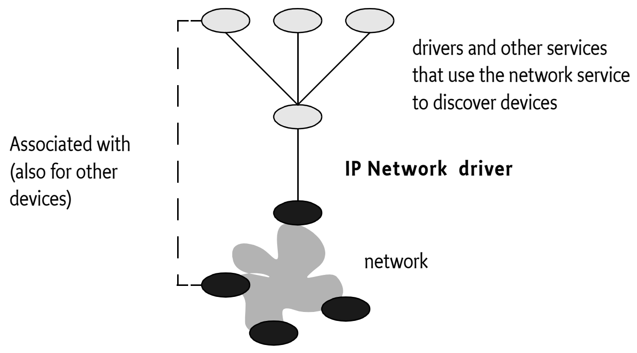
An example of a refining driver is a mouse driver, which is attached to the generic USB Device service representing a physical mouse. It then registers a new Device service which represents it as a Mouse service, defined elsewhere.

The majority of drivers fall into the refining driver type.

8.4.2.3 Network Drivers

An Internet Protocol (IP) capable network such as Ethernet supports individually addressable devices and allows broadcasts, but does not define an intrinsic discovery protocol. In this case, the entire network should be exposed as a single Device service.

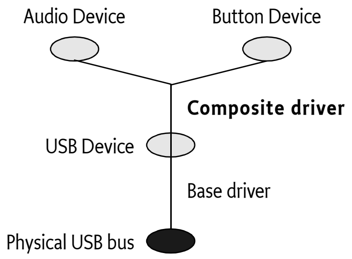
Figure 24 *Network Driver diagram*


drivers and other services that use the network service to discover devices
Associated with (also for other devices)
IP Network driver
network

8.4.2.4 Composite Drivers

Complex devices can often be broken down into several parts. Drivers that attach to a single service and then register multiple Device services are called *composite drivers*. For example, a USB speaker containing software-accessible buttons can be registered by its driver as two separate Device services: an Audio Device service and a Button Device service.

Figure 25 *Composite Driver structure*



This approach can greatly reduce the number of interfaces needed, as well as enhance reusability.

8.4.2.5 Referring Drivers

A referring driver is actually not a driver in the sense that it controls Device services. Instead, it acts as an intermediary to help locate the correct driver bundle. This process is explained in detail in *The Device Attachment Algorithm* on page 115.

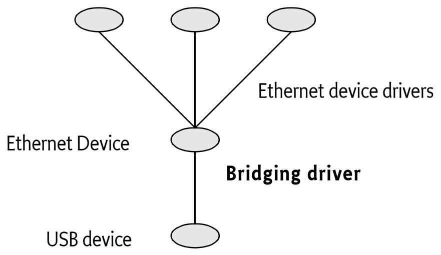
A referring driver implements the call to the `attach` method to inspect the Device service, and decides which Driver bundle would be able to attach to the device. This process can actually involve connecting to the physical device and communicating with it. The `attach` method then returns a `String` object that indicates the `DRIVER_ID` of another driver bundle. This process is called a referral.

For example, a vendor ACME can implement one driver bundle that specializes in recognizing all of the devices the vendor produces. The referring driver bundle does not contain code to control the device – it contains only sufficient logic to recognize the assortment of devices. This referring driver can be small, yet can still identify a large product line. This approach can drastically reduce the amount of downloading and matching needed to find the correct driver bundle.

8.4.2.6 Bridging Drivers

A bridging driver registers a Device service from one device category but attaches it to a Device service from another device category.

Figure 26 Bridging Driver Structure



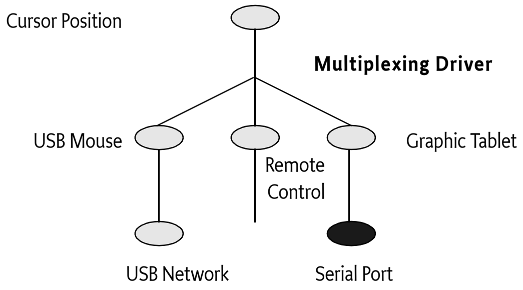
For example, USB to Ethernet bridges exist that allow connection to an Ethernet network through a USB device. In this case, the top level of the USB part of the Device service stack would be an Ethernet Device service. But the same Ethernet Device service can also be the bottom layer of an Ethernet layer of the Device service stack. A few layers up, a bridge could connect into yet another network.

The stacking depth of Device services has no limit, and the same drivers could in fact appear at different levels in the same Device service stack. The graph of drivers-to-Device services roughly mirrors the hardware connections.

8.4.2.7 Multiplexing Drivers

A *multiplexing driver* attaches a number of Device services and aggregates them in a new Device service.

Figure 27 Multiplexing Driver Structure

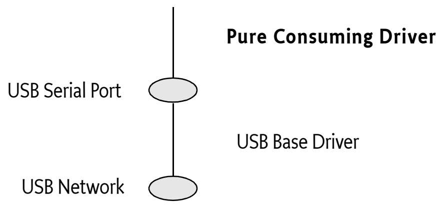


For example, assume that a system has a mouse on USB, a graphic tablet on a serial port, and a remote control facility. Each of these would be registered as a service with the Framework. A multiplexing driver can attach all three, and can merge the different positions in a central Cursor Position service.

8.4.2.8 Pure Consuming Drivers

A *pure consuming driver* bundle will attach to devices without registering a refined version.

Figure 28 Pure Consuming Driver Structure



For example, one driver bundle could decide to handle all serial ports through javax.comm instead of registering them as services. When a USB serial port is plugged in, one or more Driver services are attached, resulting in a Device service stack with a Serial Port Device service. A pure consuming driver may then attach to the Serial Port Device service and register a new serial port with the javax.comm.* registry instead of the Framework service registry. This registration effectively transfers the device from the OSGi environment into another environment.

8.4.2.9 Other Driver Types

It should be noted that any bundle installed in the OSGi environment may get and use a Device service without having to register a Driver service.

The following functionality is offered to those bundles that do register a Driver service and conform to the this specification:

- The bundles can be installed and uninstalled on demand.
- Attachment to the Device service is only initiated after the winning the competition with other drivers.

8.4.3 Driver Service Registration

Drivers are recognized by registering a Driver service with the Framework. This event makes the device manager aware of the existence of the Driver service. A Driver service registration must have a DRIVER_ID property whose value is a String object, uniquely identifying the driver to the device manager. The device manager must use the DRIVER_ID to prevent the installation of duplicate copies of the same driver bundle.

Therefore, this DRIVER_ID must:

- Depend only on the specific behavior of the driver, and thus be independent of unrelated aspects like its location or mechanism of downloading.
- Start with the reversed form of the domain name of the company that implements it: for example, com.acme.widget.1.1.

- Differ from the DRIVER_ID of drivers with different behavior. Thus, it must *also* be different for each revision of the same driver bundle so they may be distinguished.

When a new Driver service is registered, the Device Attachment Algorithm must be applied to each idle Device service. This requirement gives the new Driver service a chance to compete with other Driver services for attaching to idle devices. The techniques outlined in *Optimizations* on page 118 can provide significant shortcuts for this situation.

As a result, the Driver service object can receive match and attach requests before the method which registered the service has returned.

This specification does not define any method for new Driver services to *steal* already attached devices. Once a Device service has been attached by a Driver service, it can only be released by the Driver service itself.

8.4.4 Driver Service Unregistration

When a Driver service is unregistered, it must release all Device services to which it is attached. Thus, *all* its attached Device services become idle. The device manager must gather all of these idle Device services and try to re-attach them. This condition gives other Driver services a chance to take over the refinement of devices after the unregistering driver. The techniques outlined in *Optimizations* on page 118 can provide significant shortcuts for this situation.

A Driver service that is installed by the device manager must remain registered as long as the driver bundle is active. Therefore, a Driver service should only be unregistered if the driver bundle is stopping, an occurrence which may precede its being uninstalled or updated. Driver services should thus not unregister in an attempt to minimize resource consumption. Such optimizations can easily introduce race conditions with the device manager.

8.4.5 Driver Service Methods

The Driver interface consists of the following methods:

- match(ServiceReference) – This method is called by the device manager to find out how well this Driver service matches the Device service as indicated by the ServiceReference argument. The value returned here is specific for a device category. If this Device service is of another device category, the value Device.MATCH_NONE must be returned. Higher values indicate a better match. For the exact matching algorithm, see *The Device Attachment Algorithm* on page 115.
  Driver match values and referrals must be deterministic, in that repeated calls for the same Device service must return the same results so that results can be cached by the device manager.
- attach(ServiceReference) – If the device manager decides that a Driver service should be attached to a Device service, it must call this method on the Driver service object. Once this method is called, the Device service is regarded as attached to that Driver service, and no other Driver service must be called to attach to the Device service. The Device service must remain *owned* by the Driver service until the Driver bundle is stopped. No unattach method exists.

The attach method should return null when the Device service is correctly attached. A referring driver (see *Referring Drivers* on page 107) can return a String object that specifies the DRIVER_ID of a driver that can handle this Device service. In this case, the Device service is not attached and the device manager must attempt to install a Driver service with the same DRIVER_ID via a Driver Locator service. The attach method must be deterministic as described in the previous method.

8.4.6 Idle Driver Bundles

An idle Driver bundle is a bundle with a registered Driver service, and is not attached to any Device service. Idle Driver bundles are consuming resources in the OSGi Service Platform. The device manager should uninstall bundles that it has installed and which are idle.

8.5 Driver Locator Service

The device manager must automatically install Driver bundles, which are obtained from Driver Locator services, when new Device services are registered.

A Driver Locator service encapsulates the knowledge of how to fetch the Driver bundles needed for a specific Device service. This selection is made on the properties that are registered with a device: for example, DEVICE_CATEGORY and any other properties registered with the Device service registration.

The purpose of the Driver Locator service is to separate the mechanism from the policy. The decision to install a new bundle is made by the device manager (the mechanism), but a Driver Locator service decides which bundle to install and from where the bundle is downloaded (the policy).

Installing bundles has many consequences for the security of the system, and this process is also sensitive to network setup and other configuration details. Using Driver Locator services allows the Operator to choose a strategy that best fits its needs.

Driver services are identified by the DRIVER_ID property. Driver Locator services use this particular ID to identify the bundles that can be installed. Driver ID properties have uniqueness requirements as specified in *Device Service Registration* on page 100. This uniqueness allows the device manager to maintain a list of Driver services and prevent unnecessary installs.

An OSGi Service Platform can have several different Driver Locator services installed. The device manager must consult all of them and use the combined result set, after pruning duplicates based on the DRIVER_ID values.

8.5.1 The DriverLocator Interface

The DriverLocator interface allows suitable driver bundles to be located, downloaded, and installed on demand, even when completely unknown devices are detected.

It has the following methods:

- findDrivers(Dictionary) – This method returns an array of driver IDs that potentially match a service described by the properties in the Dictionary object. A driver ID is the String object that is registered by a Driver service under the DRIVER_ID property.
- loadDriver(String) – This method returns an InputStream object that can be used to download the bundle containing the Driver service as specified by the driver ID argument. If the Driver Locator service cannot download such a bundle, it should return null. Once this bundle is downloaded and installed in the Framework, it must register a Driver service with the DRIVER_ID property set to the value of the String argument.

8.5.2 A Driver Example

The following example shows a very minimal Driver service implementation. It consists of two classes. The first class is SerialWidget. This class tracks a single WidgetDevice from *Sample Device Category Specification* on page 102. It registers a javax.comm.SerialPort service, which is a general serial port specification that could also be implemented from other device categories like USB, a COM port, etc. It is created when the SerialWidgetDriver object is requested to attach a WidgetDevice by the device manager. It registers a new javax.comm.SerialPort service in its constructor.

The org.osgi.util.tracker.ServiceTracker is extended to handle the Framework events that are needed to simplify tracking this service. The removedService method of this class is overridden to unregister the SerialPort when the underlying WidgetDevice is unregistered.

```
package com.acme.widget;
import org.osgi.service.device.*;
import org.osgi.framework.*;
import org.osgi.util.tracker.*;

class SerialWidget extends ServiceTracker
   implements javax.comm.SerialPort,
      org.osgi.service.device.Constants {
   ServiceRegistration       registration;

   SerialWidget( BundleContext c, ServiceReference r ) {
      super( c, r, null );
      open();
   }

   public Object addingService( ServiceReference ref ) {
      WidgetDevice dev = (WidgetDevice)
         context.getService( ref );
      registration = context.registerService(
            javax.comm.SerialPort.class.getName(),
            this,
            null );
         return dev;
   }

   public void removedService( ServiceReference ref,
         Object service ) {
         registration.unregister();
         context.ungetService(ref);
      }
      ... methods for javax.comm.SerialPort that are
      ... converted to underlying WidgetDevice
   }
```

A SerialWidgetDriver object is registered with the Framework in the Bundle Activator start method under the Driver interface. The device manager must call the match method for each idle Device service that is registered. If it is chosen by the device manager to control this Device service, a new SerialWidget is created that offers serial port functionality to other bundles.

```
public class SerialWidgetDriver implements Driver {
   BundleContext           context;

   String     spec =
         "(&"
      +" (objectclass=com.acme.widget.WidgetDevice)"
      +"  (DEVICE_CATEGORY=WidgetDevice)"
      +"  (com.acme.class=Serial)"
      + ")";

   Filter     filter;

   SerialWidgetDriver( BundleContext context )
      throws Exception {
      this.context = context;
      filter = context.createFilter(spec);
   }
   public int match( ServiceReference d ) {
      if ( filter.match( d ) )
         return WidgetDevice.MATCH_CLASS;
      else
         return Device.MATCH_NONE;
   }
   public synchronized String attach(ServiceReference r){
      new SerialWidget( context, r );
   }
}
```

8.6 The Driver Selector Service

The purpose of the Driver Selector service is to customize the selection of the best Driver service from a set of suitable Driver bundles. The device manager has a default algorithm as described in *The Device Attachment Algorithm* on page 115. When this algorithm is not sufficient and requires customizing by the operator, a bundle providing a Driver Selector service can be installed in the Framework. This service must be used by the device manager as the final arbiter when selecting the best match for a Device service.

The Driver Selector service is a singleton; only one such service is recognized by the device manager. The Framework method `BundleContext.getServiceReference` must be used to obtain a Driver Selector service. In the erroneous case that multiple Driver Selector services are registered, the `service.ranking` property will thus define which service is actually used.

A device manager implementation must invoke the method `select(ServiceReference,Match[])`. This method receives a Service Reference to the Device service and an array of `Match` objects. Each `Match` object contains a link to the `ServiceReference` object of a Driver service and the result of the match value returned from a previous call to `Driver.match`. The Driver Selector service should inspect the array of `Match` objects and use some means to decide which Driver service is best suited. The index of the best match should be returned. If none of the `Match` objects describe a possible Driver service, the implementation must return `DriverSelector.SELECT_NONE` (-1).

8.7 Device Manager

Device Access is controlled by the device manager in the background. The device manager is responsible for initiating all actions in response to the registration, modification, and unregistration of Device services and Driver services, using Driver Locator services and a Driver Selector service as helpers.

The device manager detects the registration of Device services and coordinates their attachment with a suitable Driver service. Potential Driver services do not have to be active in the Framework to be eligible. The device manager must use Driver Locator services to find bundles that might be suitable for the detected Device service and that are not currently installed. This selection is done via a `DRIVER_ID` property that is unique for each Driver service.

The device manager must install and start these bundles with the help of a Driver Locator service. This activity must result in the registration of one or more Driver services. All available Driver services, installed by the device manager and also others, then participate in a bidding process. The Driver service can inspect the Device service through its `ServiceReference` object to find out how well this Driver service matches the Device service.

If a Driver Selector service is available in the Framework service registry, it is used to decide which of the eligible Driver services is the best match.

If no Driver Selector service is available, the highest bidder must win, with tie breaks defined on the `service.ranking` and `service.id` properties. The selected Driver service is then asked to `attach` the Device service.

If no Driver service is suitable, the Device service remains idle. When new Driver bundles are installed, these idle Device services must be reattached.

The device manager must reattach a Device service if, at a later time, a Driver service is unregistered due to an uninstallation or update. At the same time, however, it should prevent superfluous and non-optimal reattachments. The device manager should also garbage-collect driver bundles it installed which are no longer used.

The device manager is a singleton. Only one device manager may exist, and it must have no public interface.

8.7.1 Device Manager Startup

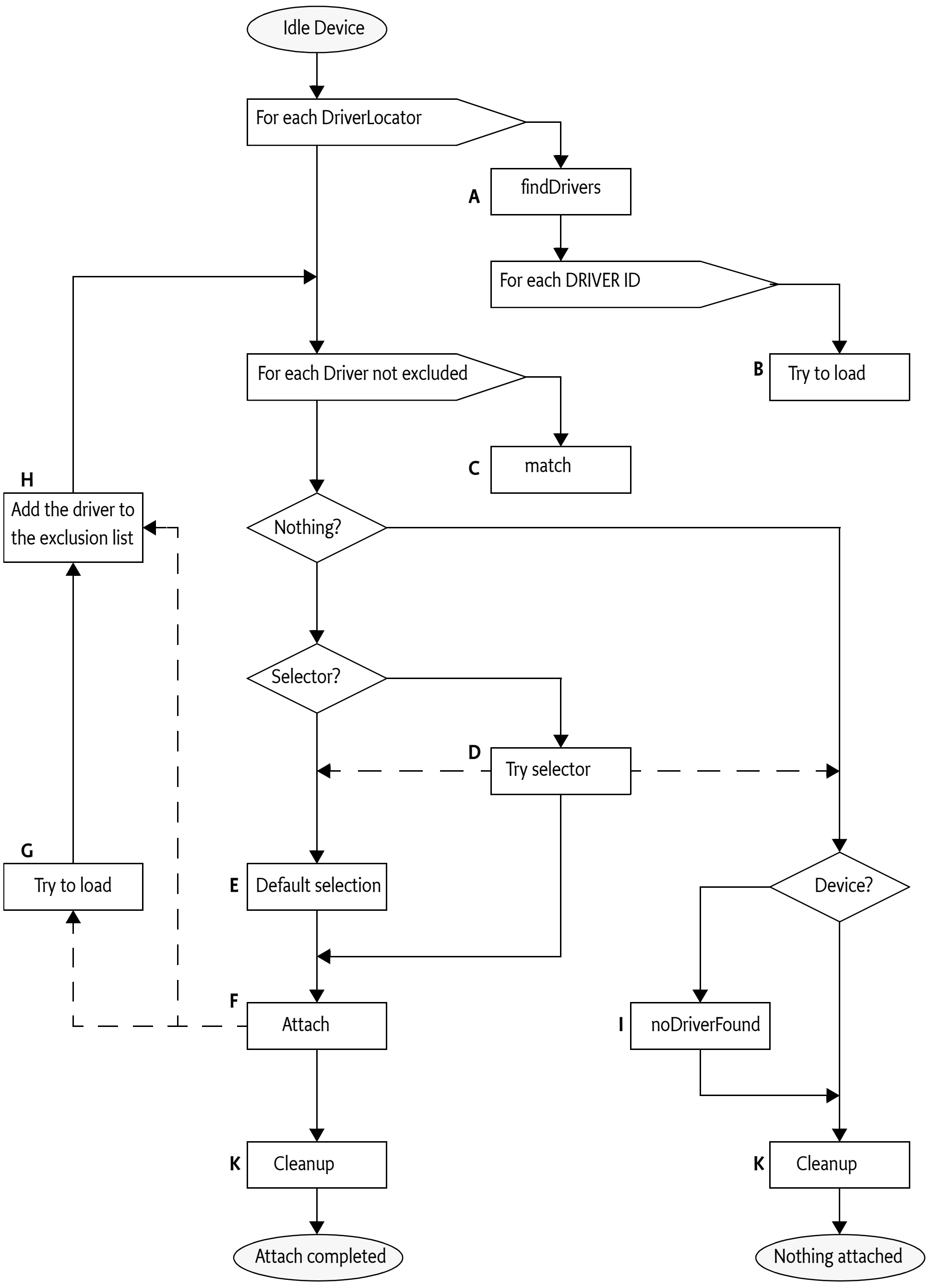
To prevent race conditions during Framework startup, the device manager must monitor the state of Device services and Driver services immediately when it is started. The device manager must not, however, begin attaching Device services until the Framework has been fully started, to prevent superfluous or non-optimal attachments.

The Framework has completed starting when the `FrameworkEvent.STARTED` event has been published. Publication of that event indicates that Framework has finished all its initialization and all bundles are started. If the device manager is started after the Framework has been initialized, it should detect the state of the Framework by examining the state of the system bundle.

8.7.2 The Device Attachment Algorithm

A key responsibility of the device manager is to attach refining drivers to idle devices. The following diagram illustrates the device attachment algorithm.

Figure 29 *Device Attachment Algorithm*



8.7.3 Legend

Step	Description
A	DriverLocator.findDrivers is called for each registered Driver Locator service, passing the properties of the newly detected Device service. Each method call returns zero or more DRIVER_ID values (identifiers of particular driver bundles). If the findDrivers method throws an exception, it is ignored, and processing continues with the next Driver Locator service. See *Optimizations* on page 118 for further guidance on handling exceptions.
B	For each found DRIVER_ID that does not correspond to an already registered Driver service, the device manager calls DriverLocator.loadDriver to return an InputStream containing the driver bundle. Each call to loadDriver is directed to one of the Driver Locator services that mentioned the DRIVER_ID in step A. If the loadDriver method fails, the other Driver Locator objects are tried. If they all fail, the driver bundle is ignored. If this method succeeds, the device manager installs and starts the driver bundle. Driver bundles must register their Driver services synchronously during bundle activation.
C	For each Driver service, except those on the exclusion list, call its Driver.match method, passing the ServiceReference object to the Device service. Collect all successful matches – that is, those whose return values are greater than Device.MATCH_NONE – in a list of active matches. A match call that throws an exception is considered unsuccessful and is not added to the list.
D	If there is a Driver Selector service, the device manager calls the DriverSelector.select method, passing the array of active Match objects. If the Driver Selector service returns the index of one of the Match objects from the array, its associated Driver service is selected for attaching the Device service. If the Driver Selector service returns DriverSelector.SELECT_NONE, no Driver service must be considered for attaching the Device service. If the Driver Selector service throws an exception or returns an invalid result, the default selection algorithm is used. Only one Driver Selector service is used, even if there is more than one registered in the Framework. See *The Driver Selector Service* on page 113.
E	The winner is the one with the highest match value. Tie breakers are respectively: • Highest service.ranking property. • Lowest service.id property.

Table 11 Driver attachment algorithm

Step	Description
F	The selected Driver service's `attach` method is called. If the `attach` method returns `null`, the Device service has been successfully attached. If the attach method returns a `String` object, it is interpreted as a referral to another Driver service and processing continues at G. See *Referring Drivers* on page 107. If an exception is thrown, the Driver service has failed, and the algorithm proceeds to try another Driver service after excluding this one from further consideration at Step H.
G	The device manager attempts to load the referred driver bundle in a manner similar to Step B, except that it is unknown which Driver Locator service to use. Therefore, the `loadDriver` method must be called on each Driver Locator service until one succeeds (or they all fail). If one succeeds, the device manager installs and starts the driver bundle. The driver bundle must register a Driver service during its activation which must be added to the list of Driver services in this algorithm.
H	The referring driver bundle is added to the exclusion list. Because each new referral adds an entry to the exclusion list, which in turn disqualifies another driver from further matching, the algorithm cannot loop indefinitely. This list is maintained for the duration of this algorithm. The next time a new Device service is processed, the exclusion list starts out empty.
I	If no Driver service attached the Device service, the Device service is checked to see whether it implements the `Device` interface. If so, the `noDriverFound` method is called. Note that this action may cause the Device service to unregister and possibly a new Device service (or services) to be registered in its place. Each new Device service registration must restart the algorithm from the beginning.
K	Whether an attachment was successful or not, the algorithm may have installed a number of driver bundles. The device manager should remove any idle driver bundles that it installed.

Table 11 Driver attachment algorithm

8.7.4 Optimizations

Optimizations are explicitly allowed and even recommended for an implementation of a device manager. Implementations may use the following assumptions:

- Driver match values and referrals must be deterministic, in that repeated calls for the same Device service must return the same results.
- The device manager may cache match values and referrals. Therefore, optimizations in the device attachment algorithm based on this assumption are allowed.
- The device manager may delay loading a driver bundle until it is needed. For example, a delay could occur when that `DRIVER_ID`'s match values are cached.

- The results of calls to DriverLocator and DriverSelector methods are not required to be deterministic, and must not be cached by the device manager.
- Thrown exceptions must not be cached. Exceptions are considered transient failures, and the device manager must always retry a method call even if it has thrown an exception on a previous invocation with the same arguments.

8.7.5 Driver Bundle Reclamation

The device manager may remove driver bundles it has installed at any time, provided that all the Driver services in that bundle are idle. This recommended practice prevents unused driver bundles from accumulating over time. Removing driver bundles too soon, however, may cause unnecessary installs and associated delays when driver bundles are needed again.

If a device manager implements driver bundle reclamation, the specified matching algorithm is not guaranteed to terminate unless the device manager takes reclamation into account.

For example, assume that a new Device service triggers the attachment algorithm. A driver bundle recommended by a Driver Locator service is loaded. It does not match, so the Device service remains idle. The device manager is eager to reclaim space, and unloads the driver bundle. The disappearance of the Driver service causes the device manager to reattach idle devices. Because it has not kept a record of its previous activities, it tries to reattach the same device, which closes the loop.

On systems where the device manager implements driver bundle reclamation, all refining drivers should be loaded through Driver Locator services. This recommendation is intended to prevent the device manager from erroneously uninstalling pre-installed driver bundles that cannot later be reinstalled when needed.

The device manager can be updated or restarted. It cannot, however, rely on previously stored information to determine which driver bundles were pre-installed and which were dynamically installed and thus are eligible for removal. The device manager may persistently store cachable information for optimization, but must be able to cold start without any persistent information and still be able to manage an existing connection state, satisfying all of the requirements in this specification.

8.7.6 Handling Driver Bundle Updates

It is not straightforward to determine whether a driver bundle is being updated when the UNREGISTER event for a Driver service is received. In order to facilitate this distinction, the device manager should wait for a period of time after the unregistration for one of the following events to occur:

- A BundleEvent.UNINSTALLED event for the driver bundle.
- A ServiceEvent.REGISTERED event for another Driver service registered by the driver bundle.

If the driver bundle is uninstalled, or if neither of the above events are received within the allotted time period, the driver is assumed to be inactive. The appropriate waiting period is implementation-dependent and will vary for different installations. As a general rule, this period should be long enough to allow a driver to be stopped, updated, and restarted under normal conditions, and short enough not to cause unnecessary delays in reattaching devices. The actual time should be configurable.

8.7.7 Simultaneous Device Service and Driver Service Registration

The device attachment algorithm may discover new driver bundles that were installed outside its direct control, which requires executing the device attachment algorithm recursively. Howerver, in this case, the appearance of the new driver bundles should be queued until completion of the current device attachment algorithm.

Only one device attachment algorithm may be in progress at any moment in time.

The following example sequence illustrates this process when a Driver service is registered:

- Collect the set of all idle devices.
- Apply the device attachment algorithm to each device in the set.
- If no Driver services were registered during the execution of the device attachment algorithm, processing terminates.
- Otherwise, restart this process.

8.8 Security

The device manager is the only privileged bundle in the Device Access specification and requires the `org.osgi.AdminPermission` to install and uninstall driver bundles.

The device manager itself should be free from any knowledge of policies and should not actively set bundle permissions. Rather, if permissions must be set, it is up to the Management Agent to listen to synchronous bundle events and set the appropriate permissions.

Driver Locator services can trigger the download of any bundle, because they deliver the content of a bundle to the privileged device manager and could potentially insert a Trojan horse into the environment. Therefore, Driver Locator bundles need the `ServicePermission[REGISTER, DriverLocator]` to register Driver Locator services, and the operator should exercise prudence in assigning this `ServicePermission`.

Bundles with Driver Selector services only require `ServicePermission[REGISTER,DriverSelector]` to register the `DriverSelector` service. The Driver Selector service can play a crucial role in the selection of a suitable Driver service, but it has no means to define a specific bundle itself.

8.9 Changes

The Device Access specification has not increased its version number because no API change has been necessary. The only change to this specification has been a clarification of the concept of an idle device.

8.10 org.osgi.service.device

The OSGi Device Access Package. Specification Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.device; specification-ver-
sion=1.1
```

8.10.1 Summary

- Constants - This interface defines standard names for property keys associated with Device[p.122] and Driver[p.122] services. [p.121]
- Device - Interface for identifying device services.[p.122]
- Driver - A `Driver` service object must be registered by each Driver bundle wishing to attach to Device services provided by other drivers. [p.122]
- DriverLocator - A Driver Locator service can find and load device driver bundles given a property set. [p.124]
- DriverSelector - When the device manager detects a new Device service, it calls all registered Driver services to determine if anyone matches the Device service. [p.124]
- Match - Instances of `Match` are used in the DriverSelector.select[p.124] method to identify Driver services matching a Device service. [p.125]

8.10.2 public interface Constants

This interface defines standard names for property keys associated with Device[p.122] and Driver[p.122] services.

The values associated with these keys are of type `java.lang.String`, unless otherwise stated.

See Also `Device`[p.122] , `Driver`[p.122]

Since 1.1

8.10.2.1 public static final String DEVICE_CATEGORY = "DEVICE_CATEGORY"

Property (named "DEVICE_CATEGORY") containing a human readable description of the device categories implemented by a device. This property is of type `String[]`

Services registered with this property will be treated as devices and discovered by the device manager

8.10.2.2 public static final String DEVICE_DESCRIPTION = "DEVICE_DESCRIPTION"

Property (named "DEVICE_DESCRIPTION") containing a human readable string describing the actual hardware device.

8.10.2.3 public static final String DEVICE_SERIAL = "DEVICE_SERIAL"

Property (named "DEVICE_SERIAL") specifying a device's serial number.

8.10.2.4 public static final String DRIVER_ID = "DRIVER_ID"

Property (named "DRIVER_ID") identifying a driver.

A `DRIVER_ID` should start with the reversed domain name of the company that implemented the driver (e.g., `com. acme`), and must meet the following requirements:

- It must be independent of the location from where it is obtained.
- It must be independent of the `DriverLocator`[p.124] service that downloaded it.
- It must be unique.
- It must be different for different revisions of the same driver.

This property is mandatory, i.e., every `Driver` service must be registered with it.

8.10.3 public interface Device

Interface for identifying device services.

A service must implement this interface or use the `Constants.DEVICE_CATEGORY`[p.121] registration property to indicate that it is a device. Any services implementing this interface or registered with the `DEVICE_CATEGORY` property will be discovered by the device manager.

Device services implementing this interface give the device manager the opportunity to indicate to the device that no drivers were found that could (further) refine it. In this case, the device manager calls the `noDriverFound`[p.122] method on the `Device` object.

Specialized device implementations will extend this interface by adding methods appropriate to their device category to it.

See Also `Driver`[p.122]

8.10.3.1 public static final int MATCH_NONE = 0

Return value from `Driver.match`[p.123] indicating that the driver cannot refine the device presented to it by the device manager. The value is zero.

8.10.3.2 public void noDriverFound( )

□ Indicates to this `Device` object that the device manager has failed to attach any drivers to it.

If this `Device` object can be configured differently, the driver that registered this `Device` object may unregister it and register a different Device service instead.

8.10.4 public interface Driver

A `Driver` service object must be registered by each Driver bundle wishing to attach to Device services provided by other drivers. For each newly discovered `Device`[p.122] object, the device manager enters a bidding phase. The `Driver` object whose `match`[p.123] method bids the highest for a particular `Device` object will be instructed by the device manager to attach to the `Device` object.

See Also `Device`[p.122] , `DriverLocator`[p.124]

8.10.4.1 public String attach(ServiceReference reference) throws Exception

reference the `ServiceReference` object of the device to attach to

□ Attaches this Driver service to the Device service represented by the given `ServiceReference` object.

A return value of `null` indicates that this Driver service has successfully attached to the given Device service. If this Driver service is unable to attach to the given Device service, but knows of a more suitable Driver service, it must return the `DRIVER_ID` of that Driver service. This allows for the implementation of referring drivers whose only purpose is to refer to other drivers capable of handling a given Device service.

After having attached to the Device service, this driver may register the underlying device as a new service exposing driver-specific functionality.

This method is called by the device manager.

Returns `null` if this Driver service has successfully attached to the given Device service, or the `DRIVER_ID` of a more suitable driver

Throws `Exception` – if the driver cannot attach to the given device and does not know of a more suitable driver

8.10.4.2 public int match(ServiceReference reference) throws Exception

reference the `ServiceReference` object of the device to match

□ Checks whether this Driver service can be attached to the Device service. The Device service is represented by the given `ServiceReference` and returns a value indicating how well this driver can support the given Device service, or `Device.MATCH_NONE`[p.122] if it cannot support the given Device service at all.

The return value must be one of the possible match values defined in the device category definition for the given Device service, or `Device.MATCH_NONE` if the category of the Device service is not recognized.

In order to make its decision, this Driver service may examine the properties associated with the given Device service, or may get the referenced service object (representing the actual physical device) to talk to it, as long as it ungets the service and returns the physical device to a normal state before this method returns.

A Driver service must always return the same match code whenever it is presented with the same Device service.

The match function is called by the device manager during the matching process.

Returns value indicating how well this driver can support the given Device service, or `Device.MATCH_NONE` if it cannot support the Device service at all

Throws `Exception` – if this Driver service cannot examine the Device service

8.10.5 public interface DriverLocator

A Driver Locator service can find and load device driver bundles given a property set. Each driver is represented by a unique `DRIVER_ID`.

Driver Locator services provide the mechanism for dynamically downloading new device driver bundles into an OSGi environment. They are supplied by providers and encapsulate all provider-specific details related to the location and acquisition of driver bundles.

See Also `Driver`[p.122]

8.10.5.1 public String[] findDrivers(Dictionary props)

props the properties of the device for which a driver is sought

□ Returns an array of `DRIVER_ID` strings of drivers capable of attaching to a device with the given properties.

The property keys in the specified `Dictionary` objects are case-insensitive.

Returns array of driver `DRIVER_ID` strings of drivers capable of attaching to a Device service with the given properties, or `null` if this Driver Locator service does not know of any such drivers

8.10.5.2 public InputStream loadDriver(String id) throws IOException

id the `DRIVER_ID` of the driver that needs to be installed.

□ Get an `InputStream` from which the driver bundle providing a driver with the giving `DRIVER_ID` can be installed.

Returns An `InputStream` object from which the driver bundle can be installed or `null` if the driver with the given ID cannot be located

Throws `IOException` – the input stream for the bundle cannot be created

8.10.6 public interface DriverSelector

When the device manager detects a new Device service, it calls all registered Driver services to determine if anyone matches the Device service. If at least one Driver service matches, the device manager must choose one. If there is a Driver Selector service registered with the Framework, the device manager will ask it to make the selection. If there is no Driver Selector service, or if it returns an invalid result, or throws an `Exception`, the device manager uses the default selection strategy.

Since 1.1

8.10.6.1 public static final int SELECT_NONE = -1

Return value from `DriverSelector.select`, if no Driver service should be attached to the Device service. The value is -1.

8.10.6.2 public int select(ServiceReference reference, Match[] matches)

reference the `ServiceReference` object of the Device service.

matches the array of all non-zero matches.

□ Select one of the matching Driver services. The device manager calls this method if there is at least one driver bidding for a device. Only Driver services that have responded with nonzero (not `Device.MATCH_NONE`[p.122]) match values will be included in the list.

Returns index into the array of `Match` objects, or `SELECT_NONE` if no Driver service should be attached

8.10.7 public interface Match

Instances of `Match` are used in the `DriverSelector.select`[p.124] method to identify Driver services matching a Device service.

See Also `DriverSelector`[p.124]

Since 1.1

8.10.7.1 public ServiceReference getDriver( )

□ Return the reference to a Driver service.

Returns `ServiceReference` object to a Driver service.

8.10.7.2 public int getMatchValue( )

□ Return the match value of this object.

Returns the match value returned by this Driver service.

8.11 References

[15] *Java Communications API*
http://java.sun.com/products/javacomm

[16] *USB Specification*
http://www.usb.org/developers/data/usbspec.zip

[17] *Universal Plug and Play*
http://www.upnp.org

[18] *Jini, Service Discovery and Usage*
http://www.jini.org/resources/

[19] *Salutation, Service Discovery Protocol*
http://www.salutation.org

5 Service Component Runtime Specification

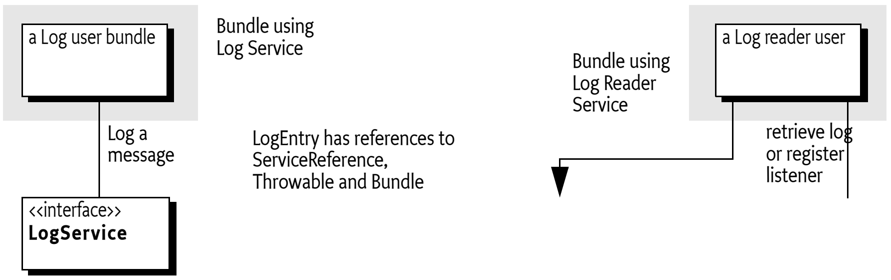
Version 1.0

5.1 Introduction

5.1.1 Entities

- *Application* – ....
- *Application Descrriptor* –

Figure 17 *Log Service Class Diagram org.osgi.service.log package*


a Log user bundle
Bundle using Log Service
Log a message
<<interface>> LogService
LogEntry has references to ServiceReference, Throwable and Bundle
Bundle using Log Reader Service
a Log reader user
retrieve log or register listener

5.2 The Service Component Runtime

5.3 Security

5.4 org.osgi.service.component

The OSGi Service Component Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.component; specification-ver-
sion=1.0
```

5.4.1 Summary

- ComponentConstants - Defines standard names for Service Component constants. [p.80]

- ComponentContext - A ComponentContext interface is used by a Service Component to interact with it execution context including locating services by reference name. [p.80]
- ComponentException - Unchecked exception which may be thrown by the Service Component Runtime. [p.82]
- ComponentFactory - When a component is declared with the `factory` attribute on it's `component` element, the Service Component Runtime will register a ComponentFactory service to allow instances of the component to be created rather than automatically create component instances as necessary. [p.82]
- ComponentInstance - A ComponentInstance encapsulates an instance of a component. [p.83]

5.4.2 public interface ComponentConstants

Defines standard names for Service Component constants.

5.4.2.1 public static final String COMPONENT_FACTORY = "component.factory"

A service registration property for a Service Component Factory. It contains the value of the `factory` attribute. The type of this property must be `String`.

5.4.2.2 public static final String COMPONENT_NAME = "component.name"

A service registration property for a Service Component. It contains the name of the Service Component. The type of this property must be `String`.

5.4.2.3 public static final String REFERENCE_TARGET_SUFFIX = ".target"

A suffix for a service registration property for a reference target. It contains the filter to select the target services for a reference. The type of this property must be `String`.

5.4.2.4 public static final String SERVICE_COMPONENT = "Service-Component"

Manifest header (named "Service-Component") identifying the XML resources within the bundle containing the bundle's Service Component descriptions.

The attribute value may be retrieved from the `Dictionary` object returned by the `Bundle.getHeaders` method.

5.4.3 public interface ComponentContext

A ComponentContext interface is used by a Service Component to interact with it execution context including locating services by reference name. In order to be notified when a component is activated and to obtain a ComponentContext, the component's implementation class must implement a

```
protected void activate(ComponentContext context);
```

method. However, the component is not required to implement this method.

In order to be called when the component is deactivated, a component's implementation class must implement a

```
protected void deactivate(ComponentContext context);
```

method. However, the component is not required to implement this method.

These methods will be called by the Service Component Runtime using reflection and may be private methods to avoid being public methods on the component's provided service object.

5.4.3.1 **public void disableComponent(String name)**

name of a component.

□ Disables the specified component name. The specified component name must be in the same bundle as this component.

5.4.3.2 **public void enableComponent(String name)**

name of a component or `null` to indicate all components in the bundle.

□ Enables the specified component name. The specified component name must be in the same bundle as this component.

5.4.3.3 **public BundleContext getBundleContext( )**

□ Returns the BundleContext of the bundle which contains this component.

Returns The BundleContext of the bundle containing this component.

5.4.3.4 **public ComponentInstance getComponentInstance( )**

□ Returns the ComponentInstance object for this component.

Returns The ComponentInstance object for this component.

5.4.3.5 **public Dictionary getProperties( )**

□ Returns the component properties for this ComponentContext.

Returns properties for this ComponentContext. The properties are read only and cannot be modified.

5.4.3.6 **public Bundle getUsingBundle( )**

□ If the component is registered as a service using the `servicefactory="true"` attribute, then this method returns the bundle using the service provided by this component.

This method will return `null` if the component is either:

- Not a service, then no bundle can be using it as a service.
- Is a service but did not specify the `servicefactory="true"` attribute, then all bundles will use this component.

Returns The bundle using this component as a service or `null`.

5.4.3.7 **public Object locateService(String name)**

name The name of a service reference as specified in a `reference` element in this component's description.

□ Returns the service object for the specified service reference name.

Returns A service object for the referenced service or `null` if the reference cardinality is `0..1` or `0..n` and no matching service is available.

Throws `ComponentException` – If the Service Component Runtime catches an exception while activating the target service.

5.4.3.8 public Object[] locateServices(String name)

name The name of a service reference as specified in a `reference` element in this component's description.

□ Returns the service objects for the specified service reference name.

Returns An array of service objects for the referenced service or `null` if the reference cardinality is `0..1` or `0..n` and no matching service is available.

Throws `ComponentException` – If the Service Component Runtime catches an exception while activating a target service.

5.4.4 public class ComponentException extends RuntimeException

Unchecked exception which may be thrown by the Service Component Runtime.

5.4.4.1 public ComponentException(String message, Throwable cause)

message The message for the exception.

cause The cause of the exception. May be `null`.

□ Construct a new ComponentException with the specified message and cause.

5.4.4.2 public ComponentException(String message)

message The message for the exception.

□ Construct a new ComponentException with the specified message.

5.4.4.3 public ComponentException(Throwable cause)

cause The cause of the exception. May be `null`.

□ Construct a new ComponentException with the specified cause.

5.4.4.4 public Throwable getCause( )

□ Returns the cause of this exception or `null` if no cause was specified when this exception was created.

Returns The cause of this exception or `null` if no cause was specified.

5.4.4.5 public Throwable initCause(Throwable cause)

□ The cause of this exception can only be set when constructed.

Throws `IllegalStateException` – This method will always throw an `IllegalStateException` since the cause of this exception can only be set when constructed.

5.4.5 public interface ComponentFactory

When a component is declared with the `factory` attribute on it's `component` element, the Service Component Runtime will register a ComponentFactory service to allow instances of the component to be created rather than automatically create component instances as necessary.

5.4.5.1 public ComponentInstance newInstance(Dictionary properties)

properties Additional properties for the component.

□ Create a new instance of the component. Additional properties may be provided for the component instance.

Returns A ComponentInstance object encapsulating the component instance. The returned component instance has been activated.

5.4.6 public interface ComponentInstance

A ComponentInstance encapsulates an instance of a component. ComponentInstances are created whenever an instance of a component is created.

5.4.6.1 public void dispose( )

□ Dispose of this component instance. The instance will be deactivated. If the instance has already been deactivated, this method does nothing.

5.4.6.2 public Object getInstance( )

□ Returns the component instance. The instance has been activated.

Returns The component instance or `null` if the instance has been deactivated.

16 Wire Admin Service Specification

Version 1.0

16.1 Introduction

The Wire Admin service is an administrative service that is used to control a wiring topology in the OSGi Service Platform. It is intended to be used by user interfaces or management programs that control the wiring of services in an OSGi Service Platform.

The Wire Admin service plays a crucial role in minimizing the amount of context-specific knowledge required by bundles when used in a large array of configurations. The Wire Admin service fulfills this role by dynamically *wiring* services together. Bundles participate in this wiring process by registering services that produce or consume data. The Wire Admin service *wires* the services that produce data to services which consume data.

The purpose of wiring services together is to allow configurable cooperation of bundles in an OSGi Service Platform. For example, a temperature sensor can be connected to a heating module to provide a controlled system.

The Wire Admin service is a very important OSGi configuration service and is designed to cooperate closely with the Configuration Admin service, as defined in *Configuration Admin Service Specification* on page 1.

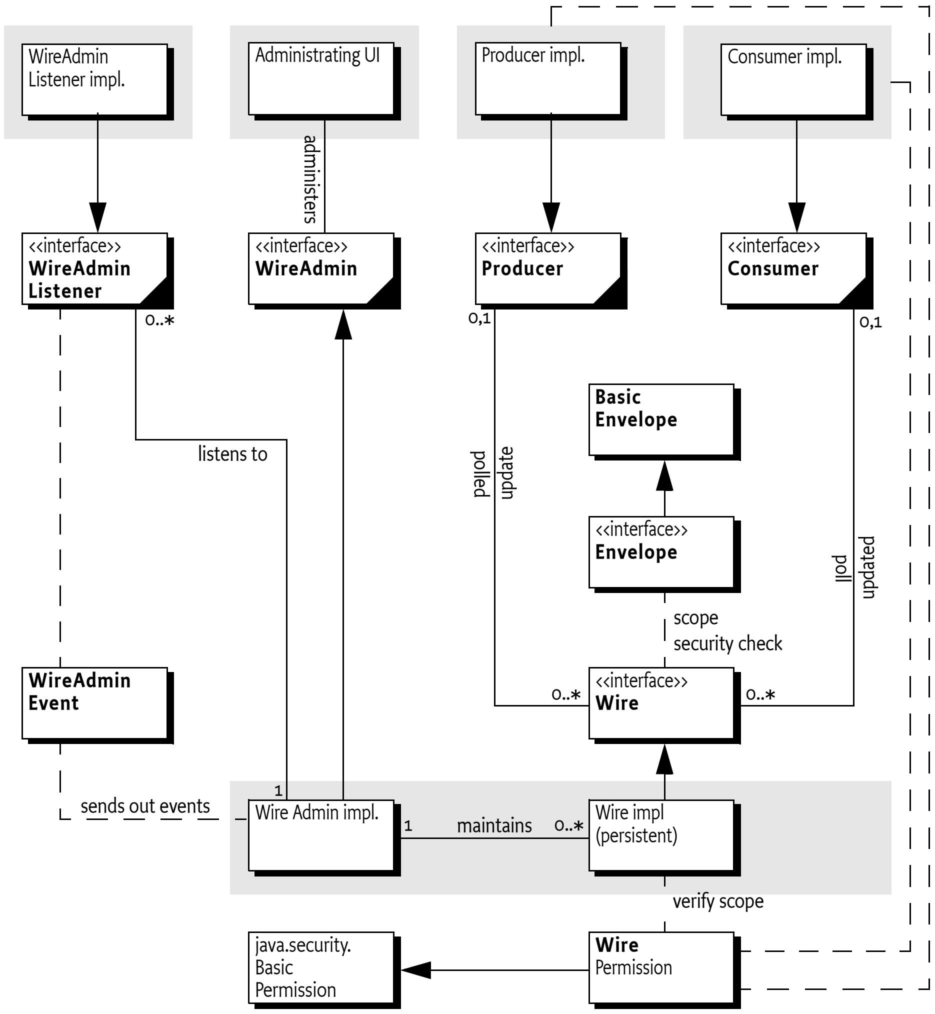
16.1.1 Wire Admin Service Essentials

- *Topology Management* – Provide a comprehensive mechanism to link data-producing components with data-consuming components in an OSGi environment.
- *Configuration Management* – Contains configuration data in order to allow either party to adapt to the special needs of the wire.
- *Data Type Handling* – Facilitate the negotiation of the data type to be used for data transfer between producers of data and consumers of data. Consumers and producers must be able to handle multiple data types for data exchanges using a preferred order.
- *Composites* – Support producers and consumers that can handle a large number of data items.
- *Security* – Separate connected parties from each other. Each party must not be required to hold the service object of the other party.
- *Simplicity* – The interfaces should be designed so that both parties, the Producer and the Consumer services, should be easy to implement.

16.1.2 Wire Admin Service Entities

- *Producer* – A service object that generates information to be used by a Consumer service.
- *Consumer* – A service object that receives information generated by a Producer service.
- *Wire* – An object created by the Wire Admin service that defines an association between a Producer service and a Consumer service. Multiple Wire objects can exist between the same Producer and Consumer pair.
- *WireAdmin* – The service that provides methods to create, update, remove, and list Wire objects.
- *WireAdminListener* – A service that receives events from the Wire Admin service when the Wire object is manipulated or used.
- *WireAdminEvent* – The event that is sent to a WireAdminListener object, describing the details of what happened.
- *Configuration Properties* – Properties that are associated with a Wire object and that contain identity and configuration information set by the administrator of the Wire Admin service.
- *PID* – The Persistent IDentity as defined in the Configuration Admin specification.
- *Flavors* – The different data types that can be used to exchange information between Producer and Consumer services.
- *Composite Producer/Consumer* – A Producer/Consumer service that can generate/accept different kinds of values.
- *Envelope* –An interface for objects that can identify a value that is transferred over the wire. Envelope objects contain also a scope name that is used to verify access permissions.
- *Scope* – A set of names that categorizes the kind of values contained in Envelope objects for security and selection purposes.
- *Basic Envelope* – A concrete implementation of the Envelope interface.
- *WirePermission* – A Permission sub-class that is used to verify if a Consumer service or Producer service has permission for specific scope names.
- *Composite Identity* – A name that is agreed between a composite Consumer and Producer service to identify the kind of objects that they can exchange.

Figure 50 *Class Diagram, org.osgi.service.wiring*



16.1.3 Operation Summary

The Wire Admin service maintains a set of persistent Wire objects. A Wire object contains a Persistent IDentity (PID) for a Consumer service and a PID for a Producer service. (Wire objects can therefore be created when the Producer or Consumer service is not registered.)

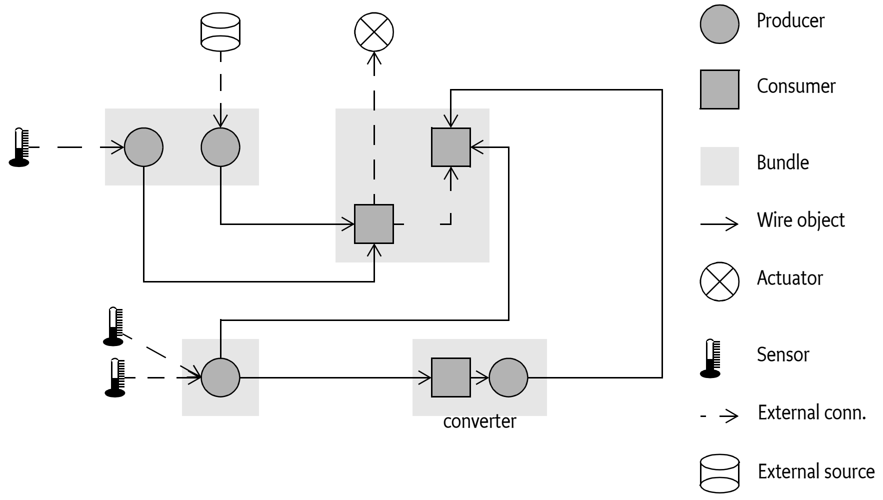
If both those Producer and Consumer services are registered with the Framework, they are connected by the Wire Admin service. The Wire Admin service calls a method on each service object and provides the list of Wire objects to which they are connected.

When a Producer service has new information, it should send this information to each of the connected Wire objects. Each Wire object then must check the filtering and security. If both filtering and security allow the transfer, the Producer service should inform the associated Consumer service with the new information. The Consumer services can also poll a Wire object for an new value at any time.

When a Consumer or Producer service is unregistered from the OSGi Framework, the other object in the association is informed that the Wire object is no longer valid.

Administrative applications can use the Wire Admin service to create and delete wires. These changes are immediately reflected in the current topology and are broadcast to Wire Admin Listener services.

Figure 51 *An Example Wiring Scheme in an OSGi Environment*



16.2 Producer Service

A Producer is a service that can produce a sequence of data objects. For example, a Producer service can produce, among others, the following type of objects:

- Measurement objects that represent a sensor measurement such as temperature, movement, or humidity.
- A String object containing information for user consumption, such as headlines.
- A Date object indicating the occurrence of a periodic event.
- Position information.
- Envelope objects containing status items which can be any type.

16.2.1 Producer Properties

A Producer service must be registered with the OSGi Framework under the interface name org.osgi.service.wireadmin.Producer. The following service properties must be set:

- service.pid – The value of this property, also known as the PID, defines the Persistent IDentity of a service. A Producer service must always use the same PID value whenever it is registered. The PID value allows the Wire Admin service to consistently identify the Producer service and create a persistent Wire object that links a Producer service to a Consumer service. See [48] *Design Patterns* specification for the rules regarding PIDs.
- wireadmin.producer.flavors – The value of this property is an array of Class objects (Class[]) that are the classes of the objects the service can produce. See *Flavors* on page 19 for more information about the data type negotiation between Producer and Consumer services.
- wireadmin.producer.filters – This property indicates to the Wire Admin service that this Producer service performs its own update filtering, meaning that the consumer can limit the number of update calls with a filter expression. This does not modify the data; it only determines whether an update via the wire occurs. If this property is not set, the Wire object must filter according to the description in *Composite objects* on page 11. This service registration property does not need to have a specific value.
- wireadmin.producer.scope – Only for a composite Producer service, a list of scope names that define the scope of this Producer service, as explained in *Scope* on page 12.
- wireadmin.producer.composite – List the composite identities of Consumer services with which this Producer service can interoperate. This property is of type String[]. A composite Consumer service can interoperate with a composite Producer service when there is at least one name that occurs in both the Consumer service's array and the Producer service's array for this property.

16.2.2 Connections

The Wire Admin service connects a Producer service and a Consumer service by creating a Wire object. If the Consumer and Producer services that are bound to a Wire object are registered with the Framework, the Wire Admin service must call the consumersConnected(Wire[]) method on the Producer service object. Every change in the Wire Admin service that affects the Wire object to which a Producer service is connected must result in a call to this method. This requirement ensures that the Producer object is informed of its role in the wiring topology. If the Producer service has no Wire objects attached when it is registered, the Wire Admin service must always call consumersConnected(null). This situation implies that a Producer service can assume it always gets called back from the Wire Admin service when it registers.

16.2.3 Producer Example

The following example shows a clock producer service that sends out a Date object every second.

```
public class Clock extends Thread implements Producer {
  Wire                wires[];
  BundleContext       context;
  boolean             quit;
```

```
    Clock( BundleContext context ) {
       this.context = context;
       start();
    }
    public synchronized void run() {
       Hashtable p = new Hashtable();
       p.put( org.osgi.service.wireadmin.WireConstants.
                   WIREADMIN_PRODUCER_FLAVORS,
                new Class[] { Date.class } );
       p.put( org.osgi.framework.Constants.SERVICE_PID,
          "com.acme.clock" );
       context.registerService(
          Producer.class.getName(),this,p );

       while( ! quit )
       try {
          Date  now = new Date();
          for( int i=0; wires!=null && i<wires.length; i++ )
               wires[i].update( now );
          wait( 1000 );
       }
       catch( InterruptedException ie) {
          /* will recheck quit */
       }
    }
    public void synchronized consumersConnected(Wire wires[])
    {
       this.wires = wires;
    }
    public Object polled(Wire wire) { return new Date(); }
   ...
}
```

16.2.4 Push and Pull

Communication between Consumer and Producer services can be initiated in one of the following ways.

- The Producer service calls the update(Object) method on the Wire object. The Wire object implementation must then call the updated(Wire,Object) method on the Consumer service, if the filtering allows this.
- The Consumer service can call poll() on the Wire object. The Wire object must then call polled(Wire) on the Producer object. Update filtering must not apply to polling.

16.2.5 Producers and Flavors

Consumer services can only understand specific data types, and are therefore restricted in what data they can process. The acceptable object classes, the flavors, are communicated by the Consumer service to the Wire Admin service using the Consumer service's service registration properties. The method getFlavors() on the Wire object returns this list of classes. This list is an ordered list in which the first class is the data type that is the most preferred data type supported by the Consumer service. The last class is the least preferred data type. The Producer service must attempt to convert its data into one of the data types according to the preferred order, or will return null from the poll method to the Consumer service if none of the types are recognized.

Classes cannot be easily compared for equivalence. Sub-classes and interfaces allow classes to masquerade as other classes. The Class.isAssignableFrom(Class) method verifies whether a class is type compatible, as in the following example:

```
Object polled(Wire wire) {
   Class clazzes[] = wire.getFlavors();
   for ( int i=0; i<clazzes.length; i++ ) {
      Class clazz = clazzes[i];
      if ( clazz.isAssignableFrom( Date.class ) )
         return new Date();
      if ( clazz.isAssignableFrom( String.class) )
         return new Date().toString();
   }
   return null;
}
```

The order of the if statements defines the preferences of the Producer object. Preferred data types are checked first. This order normally works as expected but in rare cases, sub-classes can change it. Normally, however, that is not a problem.

16.3 Consumer Service

A Consumer service is a service that receives information from one or more Producer services and is wired to Producer services by the Wire Admin service. Typical Consumer services are as follows:

- The control of an actuator, such as a heating element, oven, or electric shades
- A display
- A log
- A state controller such as an alarm system

16.3.1 Consumer Properties

A Consumer service must be registered with the OSGi Framework under the interface name org.osgi.service.wireadmin.Consumer. The following service properties must be set:

- service.pid – The value of this property, also known as the PID, defines the Persistent IDentity of a service. A Consumer service must always use the same PID value whenever it is registered. The PID value allows the Wire Admin service to consistently identify the Consumer service and create a persistent Wire object that links a Producer service to a Consumer service. See the Configuration Admin specification for the rules regarding PIDs.

- wireadmin.consumer.flavors – The value of this property is an array of Class objects (Class[]) that are the acceptable classes of the objects the service can process. See *Flavors* on page 19 for more information about the data type negotiation between Producer and Consumer services.
- wireadmin.consumer.scope – Only for a composite Consumer service, a list of scope names that define the scope of this Consumer service, as explained in *Scope* on page 12.
- wireadmin.consumer.composite – List the composite identities of Producer services that this Consumer service can interoperate with. This property is of type String[]. A composite Consumer service can interoperate with a composite Producer service when at least one name occurs in both the Consumer service's array and the Producer service's array for this property.

16.3.2 Connections

When a Consumer service is registered and a Wire object exists that associates it to a registered Producer service, the producersConnected(Wire[]) method is called on the Consumer service.

Every change in the Wire Admin service that affects a Wire object to which a Consumer service is connected must result in a call to the producersConnected(Wire[]) method. This rule ensures that the Consumer object is informed of its role in the wiring topology. If the Consumer service has no Wire objects attached, the argument to the producersConnected(Wire[]) method must be null. This method must also be called when a Producer service registers for the first time and no Wire objects are available.

16.3.3 Consumer Example

For example, a service can implement a Consumer service that logs all objects that are sent to it in order to allow debugging of a wiring topology.

```
public class LogConsumer implements Consumer {
   public LogConsumer( BundleContext context ) {
      Hashtable    ht = new Hashtable();
      ht.put(
         Constants.SERVICE_PID, "com.acme.logconsumer" );
      ht.put( WireConstants.WIREADMIN_CONSUMER_FLAVORS,
         new Class[] { Object.class } );
      context.registerService( Consumer.class.getName(),
         this, ht );
   }
   public void updated( Wire wire, Object o ) {
      getLog().log( LogService.LOG_INFO, o.toString() );
   }
   public void producersConnected( Wire [] wires) {}
   LogService getLog() { ... }
}
```

16.3.4 Polling or Receiving a Value

When the Producer service produces a new value, it calls the update(Object) method on the Wire object, which in turn calls the updated(Wire,Object) method on the Consumer service object. When the Consumer service needs a value immediately, it can call the poll() method on the Wire object which in turn calls the polled(Wire) method on the Producer service.

If the poll() method on the Wire object is called and the Producer is unregistered, it must return a null value.

16.3.5 Consumers and Flavors

Producer objects send objects of different data types through Wire objects. A Consumer service object should offer a list of preferred data types (classes) in its service registration properties. The Producer service, however, can still send a null object or an object that is not of the preferred types. Therefore, the Consumer service must check the data type and take the appropriate action. If an object type is incompatible, then a log message should be logged to allow the operator to correct the situation.

The following example illustrates how a Consumer service can handle objects of type Date, Measurement, and String.

```
void process( Object in ) {
   if ( in instanceof Date )
      processDate( (Date) in );
   else if ( in instanceof Measurement )
      processMeasurement( (Measurement) in );
   else if ( in instanceof String )
      processString( (String) in );
   else
      processError( in );
}
```

16.4 Implementation issues

The Wire Admin service can call the consumersConnected or producersConnected methods during the registration of the Consumer or Producer service. Care should be taken in this method call so that no variables are used that are not yet set, such as the ServiceRegistration object that is returned from the registration. The same is true for the updated or polled callback because setting the Wire objects on the Producer service causes such a callback from the consumersConnected or producersConnected method.

A Wire Admin service must call the producersConnected and consumersConnected method asynchronously from the registrations, meaning that the Consumer or Producer service can use synchronized to restrict access to critical variables.

When the Wire Admin service is stopped, it must disconnect all connected consumers and producers by calling producersConnected and consumersConnected with a null for the wires parameter.

16.5 Wire Properties

A Wire object has a set of properties (a Dictionary object) that configure the association between a Consumer service and a Producer service. The type and usage of the keys, as well as the allowed types for the values are defined in *Configuration Properties* on page 8.

The Wire properties are explained in Table 20.

Constant	Description
WIREADMIN_PID	The value of this property is a unique Persistent IDentity as defined in chapter 10 *Configuration Admin Service Specification.* This PID must be automatically created by the Wire Admin service for each new Wire object.
WIREADMIN_PRODUCER_PID	The value of the property is the PID of the Producer service.
WIREADMIN_CONSUMER_PID	The value of this property is the PID of the Consumer service.
WIREADMIN_FILTER	The value of this property is an OSGi filter string that is used to control the update of produced values. This filter can contain a number of attributes as explained in *Wire Flow Control* on page 15.

Table 20 Standard Wire Properties

The properties associated with a Wire object are not limited to the ones defined in Table 20. The Dictionary object can also be used for configuring *both* Consumer services and Producer services. Both services receive the Wire object and can inspect the properties and adapt their behavior accordingly.

16.5.1 Display Service Example

In the following example, the properties of a Wire object, which are set by the Operator or User, are used to configure a Producer service that monitors a user's email account regularly and sends a message when the user has received email. This WireMail service is illustrated as follows:

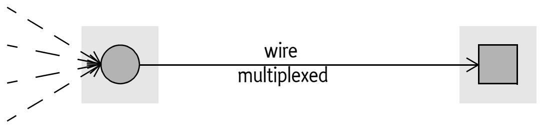
```
public class WireMail extends Thread
  implements Producer {
  Wire                wires[];
  BundleContext       context;
  boolean             quit;

  public void start( BundleContext context ) {
    Hashtable         ht = new Hashtable();
    ht.put( Constants.SERVICE_PID, "com.acme.wiremail" );
    ht.put( WireConstants.WIREADMIN_PRODUCER_FLAVORS,
        new Class[] { Integer.class } );
    context.registerService( this,
      Producer.class.getName(),
          ht );
    }
    public synchronized void  consumersConnected(
      Wire wires[] ) {
      this.wires = wires;
    }
    public Object polled( Wire wire  ) {
      Dictionary              p = wire.getProperties();
      // The password should be
      // obtained from User Admin Service
      int n = getNrMails(
        p.get( "userid" ),
        p.get( "mailhost" ) );
      return new Integer( n );
    }
    public synchronized void run() {
      while ( !quit )
      try {
        for ( int i=0; wires != null && i<wires.length;i++ )
          wires[i].update( polled( wires[i] ) );

        wait( 150000 );
      }
      catch( InterruptedException e ) { break; }
    }
    ...
  }
```

16.6 Composite objects

A Producer and/or Consumer service for each information item is usually the best solution. This solution is not feasible, however, when there are hundreds or thousands of information items. Each registered Consumer or Producer service carries the overhead of the registration, which may overwhelm a Framework implementation on smaller platforms.

When the size of the platform is an issue, a Producer and a Consumer service should abstract a larger number of information items. These Consumer and Producer services are called *composite*.

Figure 52 *Composite Producer Example*



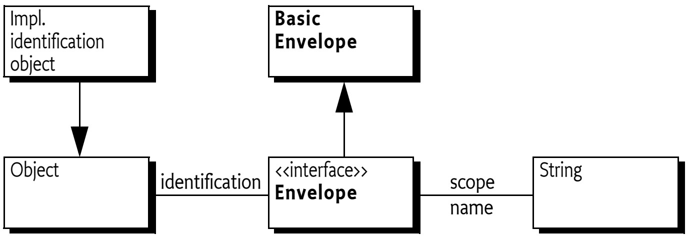
Composite Producer and Consumer services should register respectively the WIREADMIN_PRODUCER_COMPOSITE and WIREADMIN_CONSUMER_COMPOSITE *composite identity* property with their service registration. These properties should contain a list of composite identities. These identities are not defined here, but are up to a mutual agreement between the Consumer and Producer service. For example, a composite identity could be MOST-1.5 or GSM-Phase2-Terminal. The name may follow any scheme but will usually have some version information embedded. The composite identity properties are used to match Consumer and Producer services with each other during configuration of the Wire Admin service. A Consumer and Producer service should interoperate when at least one equal composite identity is listed in both the Producer and Consumer composite identity service property.

Composite producers/consumers must identify the *kind* of objects that are transferred over the Wire object, where *kind* refers to the intent of the object, not the data type. For example, a Producer service can represent the status of a door-lock and the status of a window as a boolean. If the status of the window is transferred as a boolean to the Consumer service, how would it know that this boolean represents the window and not the door-lock?

To avoid this confusion, the Wire Admin service includes an Envelope interface. The purpose of the Envelope interface is to associate a value object with:

- An identification object
- A scope name

Figure 53 *Envelope*



16.6.1 Identification

The Envelope object's identification object is used to identify the value carried in the Envelope object. Each unique kind of value must have its own unique identification object. For example, a left-front-window should have a different identification object than a rear-window.

The identification is of type Object. Using the Object class allows String objects to be used, but also makes it possible to use more complex objects. These objects can convey information in a way that is mutually agreed between the Producer and Consumer service. For example, its type may differ depending on each kind of value so that the *Visitor* pattern, see [48] *Design Patterns*, can be used. Or it may contain specific information that makes the Envelope object easier to dispatch for the Consumer service.

16.6.2 Scope

The scope name is a String object that *categorizes* the Envelope object. The scope name is used to limit the kind of objects that can be exchanged between composite Producer and Consumer services, depending on security settings.

The name-space for this scope should be mutually agreed between the Consumer and Producer services a priori. For the Wire Admin service, the scope name is an opaque string, though its syntax is specified in *Scope name syntax* on page 15.

Both composite Producer and Consumer services must add a list of their supported scope names to the service registration properties. This list is called the *scope* of that service. A Consumer service must add this scope property with the name of WIREADMIN_CONSUMER_SCOPE, a Producer service must add this scope property with the name WIREADMIN_PRODUCER_SCOPE. The type of this property must be a String[] object.

Not registering this property by the Consumer or the Producer service indicates to the Wire Admin service that any Wire object connected to that service must return null for the Wire.getScope() method. This case must be interpreted by the Consumer or Producer service that no scope verification is taking place. Secure Producer services should not produce values for this Wire object and secure Consumer services should not accept values.

It is also allowed to register with a *wildcard*, indicating that all scope names are supported. In that case, the WIREADMIN_SCOPE_ALL (which is String[] { "*" }) should be registered as the scope of the service. The Wire object's scope is then fully defined by the other service connected to the Wire object.

The following example shows how a scope is registered.

```
static String [] scope = { "DoorLock", "DoorOpen", "VIN" };

public void start( BundleContext context ) {
   Dictionary   properties = new Hashtable();
   properties.put(
      WireConstants.WIREADMIN_CONSUMER_SCOPE,
      scope );
   properties.put( WireConstants.WIREADMIN_CONSUMER_PID,
      "com.acme.composite.consumer" );
    properties.put(
       WireConstants.WIREADMIN_CONSUMER_COMPOSITE,
       new String[] { "OSGiSP-R3" } );
   context.registerService( Consumer.class.getName(),
      new AcmeConsumer(),
      properties );
}
```

Both a composite Consumer and Producer service must register a scope to receive scope support from the Wire object. These two scopes must be converted into a single Wire object's scope and scope names in this list must be checked for the appropriate permissions. This resulting scope is available from the Wire.getScope() method.

If no scope is set by either the Producer or the Consumer service the result must be null. In that case, the Producer or Consumer service must assume that no security checking is in place. A secure Consumer or Producer service should then refuse to operate with that Wire object.

Otherwise, the resulting scope is the intersection of the Consumer and Producer service scope where each name in the scope, called m, must be implied by a WirePermission[CONSUME,m] of the Consumer service, and WirePermission[PRODUCE,m] of the Producer service.

If either the Producer or Consumer service has registered a wildcard scope then it must not restrict the list of the other service, except for the permission check. If both the Producer and Consumer service registered a wildcard, the resulting list must be WIREADMIN_SCOPE_ALL (String[]{"*"}).

For example, the Consumer service has registered a scope of {A,B,C} and has WirePermission[CONSUME,*]. The Producer service has registered a scope of {B,C,E} and has WirePermission[PRODUCE,C|E]. The resulting scope is then {C}. Table 21 shows this and more examples.

Cs	Cp	Ps	Pp	Wire Scope
null		null		null
{A,B,C}	*	null		null
null		{C,D,E}		null
{A,B,C}	B|C	{A,B,C}	A|B	{B}
*	*	{A,B,C}	A|B|C	{A,B,C}
*	*	*	*	{*}
{A,B,C}	A|B|C	{A,B,C}	X	{}
{A,B,C}	*	{B,C,E}	C|E	{C}

Table 21 *Examples of scope calculation. C=Consumer, P=Producer, p=WirePermission, s=scope*

The Wire object's scope must be calculated only once, when both the Producer and Consumer service become connected. When a Producer or Consumer service subsequently modifies its scope, the Wire object must *not* modify the original scope. A Consumer and a Produce service can thus assume that the scope does not change after the producersConnected method or consumersConnected method has been called.

16.6.3 Access Control

When an Envelope object is used as argument in Wire.update(Object) then the Wire object must verify that the Envelope object's scope name is included in the Wire object's scope. If this is not the case, the update must be ignored (the updated method on the Consumer service must not be called).

A composite Producer represents a number of values, which is different from a normal Producer that can always return a single object from the poll method. A composite Producer must therefore return an array of Envelope objects (Envelope[]). This array must contain Envelope objects for all the values that are in the Wire object's scope. It is permitted to return all possible values for the Producer because the Wire object must remove all Envelope objects that have a scope name not listed in the Wire object's scope.

16.6.4 Composites and Flavors

Composite Producer and Consumer services must always use a flavor of the Envelope class. The data types of the values must be associated with the scope name or identification and mutually agreed between the Consumer and Producer services.

Flavors and Envelope objects both represent categories of different values. Flavors, however, are different Java classes that represent the same kind of value. For example, the tire pressure of the left front wheel could be passed as a Float, an Integer, or a Measurement object. Whatever data type is chosen, it is still the tire pressure of the left front wheel. The Envelope object represents the kind of object, for example the right front wheel tire pressure, or the left rear wheel.

16.6.5 Scope name syntax

Scope names are normal String objects and can, in principle, contain any Unicode character. Scope names are used with the WirePermission class that extends java.security.BasicPermission. The BasicPermission class implements the implies method and performs the name matching. The wildcard matching of this class is based on the concept of names where the constituents of the name are separated with a period ('.'): for example, org.osgi.service.http.port.

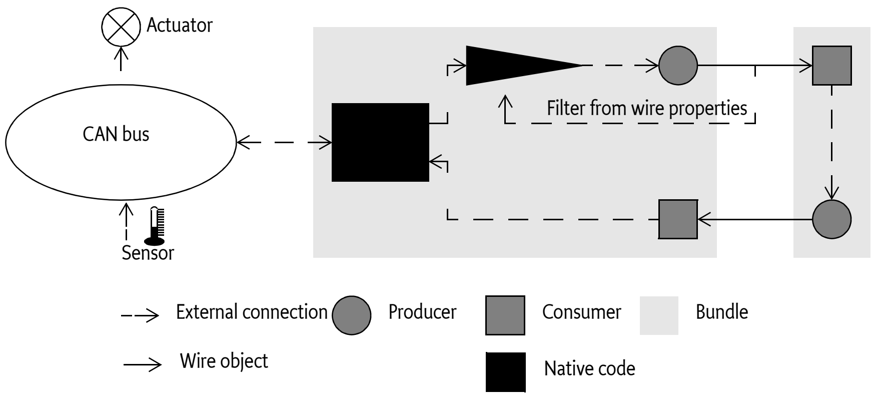
Scope names must therefore follow the rules for fully qualified Java class names. For example, door.lock is a correct scope name while door-lock is not.

16.7 Wire Flow Control

The WIREADMIN_FILTER property contains a filter expression (as defined in the OSGi Framework Filter class, see page 88) that is used to limit the number of updates to the Consumer service. This is necessary because information can arrive at a much greater rate than can be processed by a Consumer service. For example, a single CAN bus (the electronic control bus used in current cars) in a car can easily deliver hundreds of measurements per second to an OSGi based controller. Most of these measurements are not relevant to the OSGi bundles, at least not all the time. For example, a bundle that maintains an indicator for the presence of frost is only interested in measurements when the outside temperature passes the 4 degrees Celsius mark.

Limiting the number of updates from a Producer service can make a significant difference in performance (meaning that less hardware is needed). For example, a vendor can implement the filter in native code and remove unnecessary updates prior to processing in the Java Virtual Machine (JVM). This is depicted in Figure 54 on page 16.

Figure 54 *Filtering of Updates*



The filter can use any combination of the following attributes in a filter to implement many common filtering schemes:

Constant	Description
WIREVALUE_CURRENT	Current value of the data from the Producer service.
WIREVALUE_PREVIOUS	Previous data value that was reported to the Consumer service.
WIREVALUE_DELTA_ABSOLUTE	The actual positive difference between the previous data value and the current data value. For example, if the previous data value was 3 and the current data value is -0.5, then the absolute delta is 3.5. This filter attribute is not set when the current or previous value is not a number.
WIREVALUE_DELTA_RELATIVE	The absolute (meaning always positive) relative change between the current and the previous data values, calculated with the following formula: |previous-current|/|current|. For example, if the previous value was 3 and the new value is 5, then the relative delta is |3-5|/|5| = 0.4. This filter attribute is not set when the current or previous value is not a number.
WIREVALUE_ELAPSED	The time in milliseconds between the last time the Consumer.updated(Wire,Object) returned and the time the filter is evaluated.

Table 22 *Filter Attribute Names*

Filter attributes can be used to implement many common filtering schemes that limit the number of updates that are sent to a Consumer service. The Wire Admin service specification requires that updates to a Consumer service are always filtered if the WIREADMIN_FILTER Wire property is present. Producer services that wish to perform the filtering themselves should register with a service property WIREADMIN_PRODUCER_FILTERS. Filtering must be performed by the Wire object for all other Producer services.

Filtering for composite Producer services is not supported. When a filter is set on a Wire object, the Wire must still perform the filtering (which is limited to time filtering because an Envelope object is not a magnitude), but this approach may lose relevant information because the objects are of a different kind. For example, an update of every 500 ms could miss all speed updates because there is a wheel pressure update that resets the elapsed time. Producer services should, however, still implement a filtering scheme that could use proprietary attributes to filter on different kind of objects.

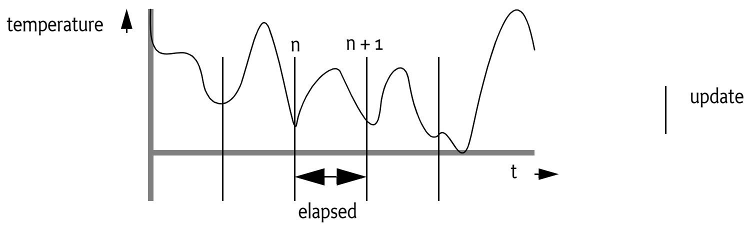
16.7.1 Filtering by Time

The simplest filter mechanism is based on time. The wirevalue.elapsed attribute contains the amount of milliseconds that have passed since the last update to the associated Consumer service. The following example filter expression illustrates how the updates can be limited to approximately 40 times per minute (once every 1500 ms).

```
(wirevalue.elapsed>=1500)
```

Figure 55 depicts this example graphically.

Figure 55 *Elapsed Time Change*



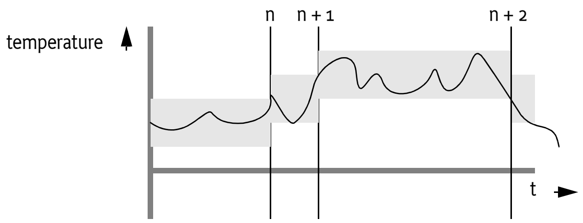
16.7.2 Filtering by Change

A Consumer service is often not interested in an update if the data value has not changed. The following filter expression shows how a Consumer service can limit the updates from a temperature sensor to be sent only when the temperature has changed at least 1 °K.

```
(wirevalue.delta.absolute>=1)
```

Figure 56 depicts a band that is created by the absolute delta between the previous data value and the current data value. The Consumer is only notified with the updated(Wire,Object) method when a data value is outside of this band.

Figure 56 *Absolute Delta*

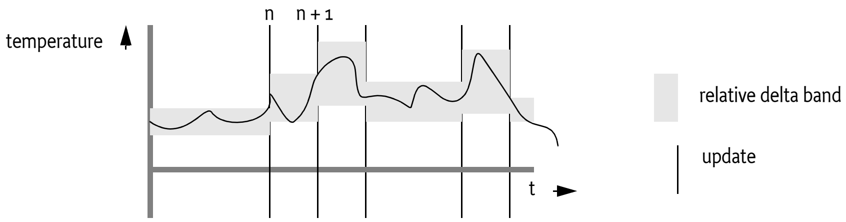


The delta may also be relative. For example, if a car is moving slowly, then updates for the speed of the car are interesting even for small variations. When a car is moving at a high rate of speed, updates are only interesting for larger variations in speed. The following example shows how the updates can be limited to data value changes of at least 10%.

```
(wirevalue.delta.relative>=0.1)
```

Figure 57 on page 18 depicts a relative band. Notice that the size of the band is directly proportional to the size of the sample value.

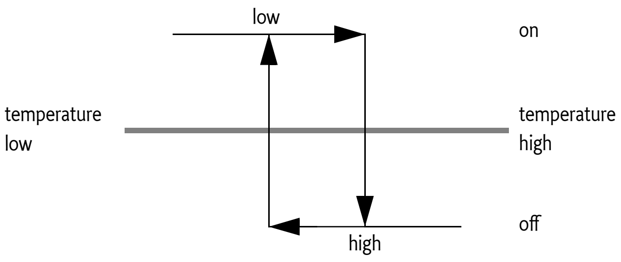
Figure 57 *Relative Delta (not on scale)*



16.7.3 Hysteresis

A thermostat is a control device that usually has a hysteresis, which means that a heater should be switched on below a certain specified low temperature and should be switched off at a specified high temperature, where *high* > *low*. This is graphically depicted in Figure 58 on page 18. The specified acceptable temperatures reduce the amount of start/stops of the heater.

Figure 58 *Hysteresis*



A Consumer service that controls the heater is only interested in events at the top and bottom of the hysteresis. If the specified high value is 250 °K and the specified low value is 249 °K, the following filter illustrates this concept:

```
(|(&(wirevalue.previous<=250)(wirevalue.current>250))
   (&(wirevalue.previous>=249)(wirevalue.current<249))
)
```

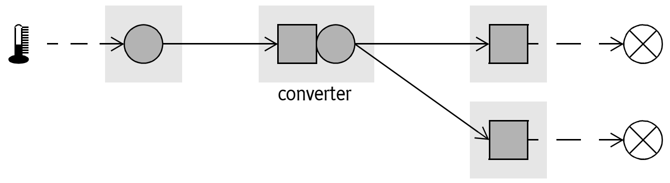
16.8 Flavors

Both Consumer and Producer services should register with a property describing the classes of the data types they can consume or produce respectively. The classes are the *flavors* that the service supports. The purpose of flavors is to allow an administrative user interface bundle to connect Consumer and Producer services. Bundles should only create a connection when there is at least one class shared between the flavors from a Consumer service and a Producer service. Producer services are responsible for selecting the preferred object type from the list of the object types preferred by the Consumer service. If the Producer service cannot convert its data to any of the flavors listed by the Consumer service, null should be used instead.

16.9 Converters

A converter is a bundle that registers a Consumer and a Producer service that are related and performs data conversions. Data values delivered to the Consumer service are processed and transferred via the related Producer service. The Producer service sends the converted data to other Consumer services. This is shown in Figure 59.

Figure 59 *Converter (for legend see Figure 51)*



16.10 Wire Admin Service Implementation

The Wire Admin service is the administrative service that is used to control the wiring topology in the OSGi Service Platform. It contains methods to create or update wires, delete wires, and list existing wires. It is intended to be used by user interfaces or management programs that control the wiring topology of the OSGi Service Platform.

The createWire(String,String,Dictionary) method is used to associate a Producer service with a Consumer service. The method always creates and returns a new object. It is therefore possible to create multiple, distinct wires between a Producer and a Consumer service. The properties can be used to create multiple associations between Producer and Consumer services in that act in different ways.

The properties of a Wire object can be updated with the update(Object) method. This method must update the properties in the Wire object and must notify the associated Consumer and Producer services if they are registered. Wire objects that are no longer needed can be removed with the deleteWire(Wire) method. All these methods are in the WireAdmin class and not in the Wire class for security reasons. See *Security* on page 23.

The getWires(String) method returns an array of Wire objects (or null). All objects are returned when the filter argument is null. Specifying a filter argument limits the returned objects. The filter uses the same syntax as the Framework Filter specification. This filter is applied to the properties of the Wire object and only Wire objects that match this filter are returned.

The following example shows how the getWires method can be used to print the PIDs of Producer services that are wired to a specific Consumer service.

```
String f = "(wireadmin.consumer.pid=com.acme.x)";
Wire [] wires = getWireAdmin().getWires( f );
for ( int i=0; wires != null && i < wires.length; i++ )
   System.out.println(
      wires[i].getProperties().get(
         "wireadmin.producer.pid")
   );
```

16.11 Wire Admin Listener Service Events

The Wire Admin service has an extensive list of events that it can deliver. The events allow other bundles to track changes in the topology as they happen. For example, a graphic user interface program can use the events to show when Wire objects become connected, when these objects are deleted, and when data flows over a Wire object.

A bundle that is interested in such events must register a WireAdminListener service object with a special Integer property WIREADMIN_EVENTS ("wireadmin.events"). This Integer object contains a bitmap of all the events in which this Wire Admin Listener service is interested (events have associated constants that can be OR'd together). A Wire Admin service must not deliver events to the Wire Admin Listener service when that event type is not in the bitmap. If no such property is registered, no events are delivered to the Wire Admin Listener service.

The WireAdminListener interface has only one method: wireAdminEvent(WireAdminEvent). The argument is a WireAdminEvent object that contains the event type and associated data.

A WireAdminEvent object can be sent asynchronously but must be ordered for each Wire Admin Listener service. Wire Admin Listener services must not assume that the state reflected by the event is still true when they receive the event.

The following types are defined for a WireEvent object:

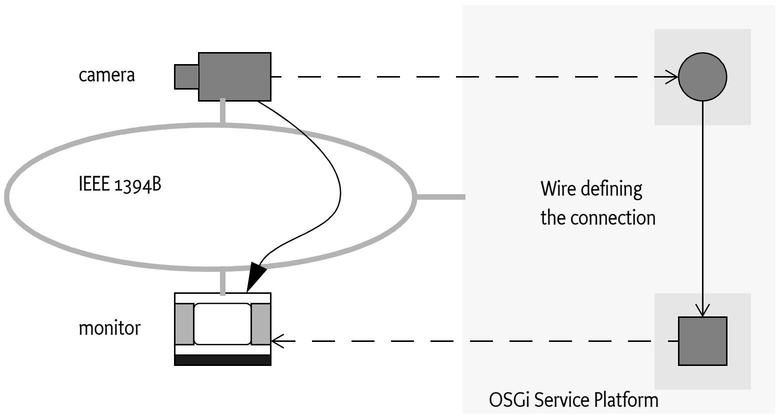
Event type	Description
WIRE_CREATED	A new Wire object has been created.
WIRE_CONNECTED	Both the Producer service and the Consumer service are registered but may not have executed their respective connectedProducers/connectedConsumers methods.
WIRE_UPDATED	The Wire object's properties have been updated.
WIRE_TRACE	The Producer service has called the Wire.update(Object) method with a new value or the Producer service has returned from the Producer.polled(Wire) method.
WIRE_DISCONNECTED	The Producer service or Consumer service have become unregistered and the Wire object is no longer connected.
WIRE_DELETED	The Wire object is deleted from the repository and is no longer available from the getWires method.
CONSUMER_EXCEPTION	The Consumer service generated an exception and the exception is included in the event.
PRODUCER_EXCEPTION	The Producer service generated an exception in a callback and the exception is included in the event.

Table 23 *Events*

16.12 Connecting External Entities

The Wire Admin service can be used to control the topology of consumers and producers that are services, as well as external entities. For example, a video camera controlled over an IEEE 1394B bus can be registered as a Producer service in the Framework's service registry and a TV, also connected to this bus, can be registered as a Consumer service. It would be very inefficient to stream the video data through the OSGi environment. Therefore, the Wire Admin service can be used to supply the external addressing information to the camera and the monitor to make a direct connection *outside* the OSGi environment. The Wire Admin service provides a uniform mechanism to connect both external entities and internal entities.

Figure 60 Connecting External Entities



A Consumer service and a Producer service associated with a Wire object receive enough information to establish a direct link because the PIDs of both services are in the Wire object's properties. This situation, however, does not guarantee *compatibility* between Producer and the Consumer service. It is therefore recommended that flavors are used to ensure this compatibility. Producer services that participate in an external addressing scheme, like IEEE 1394B, should have a flavor that reflects this address. In this case, there should then for example be a IEEE 1394B address class. Consumer services that participate in this external addressing scheme should only accept data of this flavor.

The OSGi *Device Access Specification* on page 1, defines the concept of a device category. This is a description of what classes and properties are used in a specific device category: for example, a UPnP device category that defines the interface that must be used to register for a UPnP device, among other things.

Device category descriptions should include a section that addresses the external wiring issue. This section should include what objects are send over the wire to exchange addressing information.

16.13 Related Standards

16.13.1 Java Beans

The Wire Admin service leverages the component architecture that the Framework service registry offers. Java Beans attempt to achieve similar goals. Java Beans are classes that follow a number of recommendations that allow them to be configured at run time. The techniques that are used by Java Beans during configuration are serialization and the construction of adapter classes.

Creating adapter classes in a resource constrained OSGi Service Platform was considered too heavy weight. Also, the dynamic nature of the OSGi environment, where services are registered and unregistered continuously, creates a mismatch between the intended target area of Java Beans and the OSGi Service Platform.

Also, Java Beans can freely communicate once they have a reference to each other. This freedom makes it impossible to control the communication between Java Beans.

This Wire Admin service specification was developed because it is lightweight and leverages the unique characteristics of the OSGi Framework. The concept of a Wire object that acts as an intermediate between the Producer and Consumer service allows the implementation of a security policy because both parties cannot communicate directly.

16.14 Security

16.14.1 Separation of Consumer and Producer Services

The Consumer and Producer service never directly communicate with each other. All communication takes place through a Wire object. This allows a Wire Admin service implementation to control the security aspects of creating a connection, and implies that the Wire Admin service must be a trusted service in a secure environment. Only one bundle should have the ServicePermission[REGISTER,WireAdmin].

ServicePermission[REGISTER,Producer|Consumer] should not be restricted. ServicePermission[GET,Producer|Consumer] must be limited to trusted bundles (the Wire Admin service implementation) because a bundle with this permission can call such services and access information that it should not be able to access.

16.14.2 Using Wire Admin Service

This specification assumes that only a few applications require access to the Wire Admin service. The WireAdmin interface contains all the security sensitive methods that create, update, and remove Wire objects. (This is the reason that the update and delete methods are on the WireAdmin interface and not on the Wire interface). ServicePermission[GET,WireAdmin] should therefore only be given to trusted bundles that can manage the topology.

16.14.3 Wire Permission

Composite Producer and Consumer services can be restricted in their use of scope names. This restriction is managed with the WirePermission class. A WirePermission consists of a scope name and the action CONSUME or PRODUCE. The name used with the WirePermission may contain wild-cards as specified in the java.security.BasicPermission class.

16.15 org.osgi.service.wireadmin

The OSGi Wire Admin service Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.wireadmin; specification-ver-
sion=1.0
```

16.15.1 Summary

- BasicEnvelope - `BasicEnvelope` is an implementation of the `Envelope`[p.26] interface [p.24]
- Consumer - Data Consumer, a service that can receive udpated values from `Producer`[p.26] services. [p.24]
- Envelope - Identifies a contained value. [p.26]
- Producer - Data Producer, a service that can generate values to be used by `Consumer`[p.24] services. [p.26]
- Wire - A connection between a Producer service and a Consumer service. [p.28]
- WireAdmin - Wire Administration service. [p.32]
- WireAdminEvent - A Wire Admin Event. [p.33]
- WireAdminListener - Listener for Wire Admin Events. [p.36]
- WireConstants - Defines standard names for `Wire` properties, wire filter attributes, Consumer and Producer service properties. [p.37]
- WirePermission - Permission for the scope of a `Wire` object. [p.40]

16.15.2 public class BasicEnvelope implements Envelope

`BasicEnvelope` is an implementation of the `Envelope`[p.26] interface

16.15.2.1 public BasicEnvelope(Object value, Object identification, String scope)

value Content of this envelope, may be `null`.

identification Identifying object for this `Envelope` object, must not be `null`

scope Scope name for this object, must not be `null`

□ Constructor.

See Also `Envelope`[p.26]

16.15.2.2 public Object getIdentification( )

See Also `org.osgi.service.wireadmin.Envelope.getIdentification()`[p.26]

16.15.2.3 public String getScope( )

See Also `org.osgi.service.wireadmin.Envelope.getScope()`[p.26]

16.15.2.4 public Object getValue( )

See Also `org.osgi.service.wireadmin.Envelope.getValue()`[p.26]

16.15.3 public interface Consumer

Data Consumer, a service that can receive udpated values from `Producer`[p.26] services.

Service objects registered under the `Consumer` interface are expected to consume values from a Producer service via a `Wire` object. A Consumer service may poll the Producer service by calling the `Wire.poll`[p.31] method. The Consumer service will also receive an updated value when called at it's `updated`[p.25] method. The Producer service should have coerced the value to be an instance of one of the types specified by the `Wire.getFlavors`[p.29] method, or one of their subclasses.

Consumer service objects must register with a `service.pid` and a `WireConstants.WIREADMIN_CONSUMER_FLAVORS`[p.37] property. It is recommended that Consumer service objects also register with a `service.description` property.

If an `Exception` is thrown by any of the `Consumer` methods, a `WireAdminEvent` of type `WireAdminEvent.CONSUMER_EXCEPTION`[p.34] is broadcast by the Wire Admin service.

Security Considerations - Data consuming bundles will require `ServicePermission[REGISTER,Consumer]`. In general, only the Wire Admin service bundle should have this permission. Thus only the Wire Admin service may directly call a Consumer service. Care must be taken in the sharing of `Wire` objects with other bundles.

Consumer services must be registered with their scope when they can receive different types of objects from the Producer service. The Consumer service should have `WirePermission` for each of these scope names.

16.15.3.1 public void producersConnected(Wire[] wires)

wires An array of the current and complete list of `Wire` objects to which this Consumer service is connected. May be `null` if the Consumer service is not currently connected to any `Wire` objects.

□ Update the list of `Wire` objects to which this Consumer service is connected.

This method is called when the Consumer service is first registered and subsequently whenever a `Wire` associated with this Consumer service becomes connected, is modified or becomes disconnected.

The Wire Admin service must call this method asynchronously. This implies that implementors of Consumer can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

16.15.3.2 public void updated(Wire wire, Object value)

wire The `Wire` object which is delivering the updated value.

value The updated value. The value should be an instance of one of the types specified by the `Wire.getFlavors`[p.29] method.

□ Update the value. This Consumer service is called by the `Wire` object with an updated value from the Producer service.

Note: This method may be called by a `Wire` object prior to this object being notified that it is connected to that `Wire` object (via the `producersConnected`[p.25] method).

When the Consumer service can receive Envelope objects, it must have registered all scope names together with the service object, and each of those names must be permitted by the bundle's WirePermission. If an Envelope object is delivered with the updated method, then the Consumer service should assume that the security check has been performed.

16.15.4 public interface Envelope

Identifies a contained value. An Envelope object combines a status value, an identification object and a scope name. The Envelope object allows the use of standard Java types when a Producer service can produce more than one kind of object. The Envelope object allows the Consumer service to recognize the kind of object that is received. For example, a door lock could be represented by a Boolean object. If the Producer service would send such a Boolean object, then the Consumer service would not know what door the Boolean object represented. The Envelope object contains an identification object so the Consumer service can discriminate between different kinds of values. The identification object may be a simple String object, but it can also be a domain specific object that is mutually agreed by the Producer and the Consumer service. This object can then contain relevant information that makes the identification easier.

The scope name of the envelope is used for security. The Wire object must verify that any Envelope object send through the update method or coming from the poll method has a scope name that matches the permissions of both the Producer service and the Consumer service involved. The wireadmin package also contains a class BasicEnvelope that implements the methods of this interface.

See Also WirePermission[p.40], BasicEnvelope[p.24]

16.15.4.1 public Object getIdentification( )

□ Return the identification of this Envelope object. An identification may be of any Java type. The type must be mutually agreed between the Consumer and Producer services.

Returns an object which identifies the status item in the address space of the composite producer, must not be null.

16.15.4.2 public String getScope( )

□ Return the scope name of this Envelope object. Scope names are used to restrict the communication between the Producer and Consumer services. Only Envelopes objects with a scope name that is permitted for the Producer and the Consumer services must be passed through a Wire object.

Returns the security scope for the status item, must not be null.

16.15.4.3 public Object getValue( )

□ Return the value associated with this Envelope object.

Returns the value of the status item, or null when no item is associated with this object.

16.15.5 public interface Producer

Data Producer, a service that can generate values to be used by Consumer[p.24] services.

Service objects registered under the Producer interface are expected to produce values (internally generated or from external sensors). The value can be of different types. When delivering a value to a Wire object, the Producer service should coerce the value to be an instance of one of the types specified by Wire.getFlavors[p.29] . The classes are specified in order of preference.

When the data represented by the Producer object changes, this object should send the updated value by calling the update method on each of Wire objects passed in the most recent call to this object's consumersConnected[p.27] method. These Wire objects will pass the value on to the associated Consumer service object.

The Producer service may use the information in the Wire object's properties to schedule the delivery of values to the Wire object.

Producer service objects must register with a service.pid and a WireConstants.WIREADMIN_PRODUCER_FLAVORS[p.39] property. It is recommended that a Producer service object also registers with a service.description property. Producer service objects must register with a WireConstants.WIREADMIN_PRODUCER_FILTERS[p.39] property if the Producer service will be performing filtering instead of the Wire object.

If an exception is thrown by a Producer object method, a WireAdminEvent of type WireAdminEvent.PRODUCER_EXCEPTION[p.34] is broadcast by the Wire Admin service.

Security Considerations. Data producing bundles will require ServicePermission[REGISTER, Producer] to register a Producer service. In general, only the Wire Admin service should have ServicePermission[GET, Producer]. Thus only the Wire Admin service may directly call a Producer service. Care must be taken in the sharing of Wire objects with other bundles.

Producer services must be registered with scope names when they can send different types of objects (composite) to the Consumer service. The Producer service should have WirePermission for each of these scope names.

16.15.5.1 public void consumersConnected(Wire[] wires)

wires An array of the current and complete list of Wire objects to which this Producer service is connected. May be null if the Producer is not currently connected to any Wire objects.

□ Update the list of Wire objects to which this Producer object is connected.

This method is called when the Producer service is first registered and subsequently whenever a Wire associated with this Producer becomes connected, is modified or becomes disconnected.

The Wire Admin service must call this method asynchronously. This implies that implementors of a Producer service can be assured that the call-back will not take place during registration when they execute the registration in a synchronized method.

16.15.5.2 public Object polled(Wire wire)

wire The `Wire` object which which is polling this service.

□ Return the current value of this `Producer` object.

This method is called by a `Wire` object in response to the Consumer service calling the `Wire` object's `poll` method. The Producer should coerce the value to be an instance of one of the types specified by `Wire.getFlavors`[p.29]. The types are specified in order of of preference. The returned value should be as new or newer than the last value furnished by this object.

Note: This method may be called by a `Wire` object prior to this object being notified that it is connected to that `Wire` object (via the `consumersConnected`[p.27] method).

If the Producer service returns an `Envelope` object that has an unpermitted scope name, then the Wire object must ignore (or remove) the transfer.

If the `Wire` object has a scope set, the return value must be an array of `Envelope` objects (`Envelope[]`). The `Wire` object must have removed any `Envelope` objects that have a scope name that is not in the Wire object's scope.

Returns The current value of the Producer service or `null` if the value cannot be coerced into a compatible type. Or an array of `Envelope` objects.

16.15.6 public interface Wire

A connection between a Producer service and a Consumer service.

A `Wire` object connects a Producer service to a Consumer service. Both the Producer and Consumer services are identified by their unique `service.pid` values. The Producer and Consumer services may communicate with each other via `Wire` objects that connect them. The Producer service may send updated values to the Consumer service by calling the `update`[p.31] method. The Consumer service may request an updated value from the Producer service by calling the `poll`[p.31] method.

A Producer service and a Consumer service may be connected through multiple `Wire` objects.

Security Considerations. `Wire` objects are available to Producer and Consumer services connected to a given `Wire` object and to bundles which can access the `WireAdmin` service. A bundle must have `ServicePermission[GET,WireAdmin]` to get the `WireAdmin` service to access all `Wire` objects. A bundle registering a Producer service or a Consumer service must have the appropriate `ServicePermission[REGISTER, Consumer|Producer]` to register the service and will be passed `Wire` objects when the service object's `consumersConnected` or `producersConnected` method is called.

Scope. Each Wire object can have a scope set with the `setScope` method. This method should be called by a Consumer service when it assumes a Producer service that is composite (supports multiple information items). The names in the scope must be verified by the `Wire` object before it is used in communication. The semantics of the names depend on the Producer service and must not be interpreted by the Wire Admin service.

16.15.6.1 public Class[] getFlavors( )

☐ Return the list of data types understood by the Consumer service connected to this `Wire` object. Note that subclasses of the classes in this list are acceptable data types as well.

The list is the value of the `WireConstants.WIREADMIN_CONSUMER_FLAVORS`[p.37] service property of the Consumer service object connected to this object. If no such property was registered or the type of the property value is not `Class[]`, this method must return `null`.

Returns An array containing the list of classes understood by the Consumer service or `null` if the `Wire` is not connected, or the consumer did not register a `WireConstants.WIREADMIN_CONSUMER_FLAVORS`[p.37] property or the value of the property is not of type `Class[]`.

16.15.6.2 public Object getLastValue( )

☐ Return the last value sent through this `Wire` object.

The returned value is the most recent, valid value passed to the `update`[p.31] method or returned by the `poll`[p.31] method of this object. If filtering is performed by this `Wire` object, this methods returns the last value provided by the Producer service. This value may be an `Envelope[]` when the Producer service uses scoping. If the return value is an Envelope object (or array), it must be verified that the Consumer service has the proper WirePermission to see it.

Returns The last value passed though this `Wire` object or `null` if no valid values have been passed or the Consumer service has no permission.

16.15.6.3 public Dictionary getProperties( )

☐ Return the wire properties for this `Wire` object.

Returns The properties for this `Wire` object. The returned `Dictionary` must be read only.

16.15.6.4 public String[] getScope( )

☐ Return the calculated scope of this `Wire` object. The purpose of the `Wire` object's scope is to allow a Producer and/or Consumer service to produce/consume different types over a single `Wire` object (this was deemed necessary for efficiency reasons). Both the Consumer service and the Producer service must set an array of scope names (their scope) with the service registration property `WIREADMIN_PRODUCER_SCOPE`, or `WIREADMIN_CONSUMER_SCOPE` when they can produce multiple types. If a Producer service can produce different types, it should set this property to the array of scope names it can produce, the Consumer service must set the array of scope names it can consume. The scope of a `Wire` object is defined as the intersection of permitted scope names of the Producer service and Consumer service.

If neither the Consumer, or the Producer service registers scope names with its service registration, then the `Wire` object's scope must be `null`.

The `Wire` object's scope must not change when a Producer or Consumer services modifies its scope.

A scope name is permitted for a Producer service when the registering bundle has `WirePermission[PRODUCE]`, and for a Consumer service when the registering bundle has `WirePermission[CONSUME]`.

If either Consumer service or Producer service has not set a `WIREADMIN_*_SCOPE` property, then the returned value must be `null`.

If the scope is set, the `Wire` object must enforce the scope names when `Envelope` objects are used as a parameter to update or returned from the `poll` method. The `Wire` object must then remove all `Envelope` objects with a scope name that is not permitted.

Returns A list of permitted scope names or null if the Produce or Consumer service has set no scope names.

16.15.6.5 public boolean hasScope(String name)

name The scope name

□ Return true if the given name is in this `Wire` object's scope.

Returns true if the name is listed in the permitted scope names

16.15.6.6 public boolean isConnected( )

□ Return the connection state of this `Wire` object.

A `Wire` is connected after the Wire Admin service receives notification that the Producer service and the Consumer service for this `Wire` object are both registered. This method will return `true` prior to notifying the Producer and Consumer services via calls to their respective `consumersConnected` and `producersConnected` methods.

A `WireAdminEvent` of type `WireAdminEvent.WIRE_CONNECTED`[p.34] must be broadcast by the Wire Admin service when the `Wire` becomes connected.

A `Wire` object is disconnected when either the Consumer or Producer service is unregistered or the `Wire` object is deleted.

A `WireAdminEvent` of type `WireAdminEvent.WIRE_DISCONNECTED`[p.35] must be broadcast by the Wire Admin service when the `Wire` becomes disconnected.

Returns `true` if both the Producer and Consumer for this `Wire` object are connected to the `Wire` object; `false` otherwise.

16.15.6.7 public boolean isValid( )

□ Return the state of this `Wire` object.

A connected `Wire` must always be disconnected before becoming invalid.

Returns `false` if this `Wire` object is invalid because it has been deleted via `WireAdmin.deleteWire`[p.33] ; `true` otherwise.

16.15.6.8 public Object poll( )

□ Poll for an updated value.

This methods is normally called by the Consumer service to request an updated value from the Producer service connected to this `Wire` object. This `Wire` object will call the `Producer.polled`[p.28] method to obtain an updated value. If this `Wire` object is not connected, then the Producer service must not be called.

If this `Wire` object has a scope, then this method must return an array of `Envelope` objects. The objects returned must match the scope of this object. The `Wire` object must remove all `Envelope` objects with a scope name that is not in the `Wire` object's scope. Thus, the list of objects returned must only contain `Envelope` objects with a permitted scope name. If the array becomes empty, `null` must be returned.

A `WireAdminEvent` of type `WireAdminEvent.WIRE_TRACE`[p.35] must be broadcast by the Wire Admin service after the Producer service has been successfully called.

Returns A value whose type should be one of the types returned by `getFlavors`[p.29] , `Envelope []`, or `null` if the `Wire` object is not connected, the Producer service threw an exception, or the Producer service returned a value which is not an instance of one of the types returned by `getFlavors`[p.29] .

16.15.6.9 public void update(Object value)

value The updated value. The value should be an instance of one of the types returned by `getFlavors`[p.29] .

□ Update the value.

This methods is called by the Producer service to notify the Consumer service connected to this `Wire` object of an updated value.

If the properties of this `Wire` object contain a `WireConstants.WIREADMIN_FILTER`[p.38] property, then filtering is performed. If the Producer service connected to this `Wire` object was registered with the service property `WireConstants.WIREADMIN_PRODUCER_FILTERS`[p.39] , the Producer service will perform the filtering according to the rules specified for the filter. Otherwise, this `Wire` object will perform the filtering of the value.

If no filtering is done, or the filter indicates the updated value should be delivered to the Consumer service, then this `Wire` object must call the `Consumer.updated`[p.25] method with the updated value. If this `Wire` object is not connected, then the Consumer service must not be called and the value is ignored.

If the value is an `Envelope` object, and the scope name is not permitted, then the `Wire` object must ignore this call and not transfer the object to the Consumer service.

A `WireAdminEvent` of type `WireAdminEvent.WIRE_TRACE`[p.35] must be broadcast by the Wire Admin service after the Consumer service has been successfully called.

See Also WireConstants.WIREADMIN_FILTER[p.38]

16.15.7 public interface WireAdmin

Wire Administration service.

This service can be used to create Wire objects connecting a Producer service and a Consumer service. Wire objects also have wire properties that may be specified when a Wire object is created. The Producer and Consumer services may use the Wire object's properties to manage or control their interaction. The use of Wire object's properties by a Producer or Consumer services is optional.

Security Considerations. A bundle must have ServicePermission[GET, WireAdmin] to get the Wire Admin service to create, modify, find, and delete Wire objects.

16.15.7.1 public Wire createWire(String producerPID, String consumerPID, Dictionary properties)

producerPID The service.pid of the Producer service to be connected to the Wire object.

consumerPID The service.pid of the Consumer service to be connected to the Wire object.

properties The Wire object's properties. This argument may be null if the caller does not wish to define any Wire object's properties.

□ Create a new Wire object that connects a Producer service to a Consumer service. The Producer service and Consumer service do not have to be registered when the Wire object is created.

The Wire configuration data must be persistently stored. All Wire connections are reestablished when the WireAdmin service is registered. A Wire can be permanently removed by using the deleteWire[p.33] method.

The Wire object's properties must have case insensitive String objects as keys (like the Framework). However, the case of the key must be preserved.

The WireAdmin service must automatically add the following Wire properties:

- WireConstants.WIREADMIN_PID[p.39] set to the value of the Wire object's persistent identity (PID). This value is generated by the Wire Admin service when a Wire object is created.
- WireConstants.WIREADMIN_PRODUCER_PID[p.39] set to the value of Producer service's PID.
- WireConstants.WIREADMIN_CONSUMER_PID[p.38] set to the value of Consumer service's PID.

If the properties argument already contains any of these keys, then the supplied values are replaced with the values assigned by the Wire Admin service.

The Wire Admin service must broadcast a WireAdminEvent of type WireAdminEvent.WIRE_CREATED[p.35] after the new Wire object becomes available from getWires[p.33] .

Returns The Wire object for this connection.

Throws IllegalArgumentException – If properties contains invalid wire types or case variants of the same key name.

16.15.7.2 public void deleteWire(Wire wire)

wire The Wire object which is to be deleted.

□ Delete a Wire object.

The Wire object representing a connection between a Producer service and a Consumer service must be removed. The persistently stored configuration data for the Wire object must destroyed. The Wire object's method Wire.isValid[p.30] will return false after it is deleted.

The Wire Admin service must broadcast a WireAdminEvent of type WireAdminEvent.WIRE_DELETED[p.35] after the Wire object becomes invalid.

16.15.7.3 public Wire[] getWires(String filter) throws InvalidSyntaxException

filter Filter string to select Wire objects or null to select all Wire objects.

□ Return the Wire objects that match the given filter.

The list of available Wire objects is matched against the specified filter.Wire objects which match the filter must be returned. These Wire objects are not necessarily connected. The Wire Admin service should not return invalid Wire objects, but it is possible that a Wire object is deleted after it was placed in the list.

The filter matches against the Wire object's properties including WireConstants.WIREADMIN_PRODUCER_PID[p.39] , WireConstants.WIREADMIN_CONSUMER_PID[p.38] and WireConstants.WIREADMIN_PID[p.39] .

Returns An array of Wire objects which match the filter or null if no Wire objects match the filter.

Throws InvalidSyntaxException – If the specified filter has an invalid syntax.

See Also org.osgi.framework.Filter

16.15.7.4 public void updateWire(Wire wire, Dictionary properties)

wire The Wire object which is to be updated.

properties The new Wire object's properties or null if no properties are required.

□ Update the properties of a Wire object. The persistently stored configuration data for the Wire object is updated with the new properties and then the Consumer and Producer services will be called at the respective Consumer.producersConnected[p.25] and Producer.consumersConnected[p.27] methods.

The Wire Admin service must broadcast a WireAdminEvent of type WireAdminEvent.WIRE_UPDATED[p.35] after the updated properties are available from the Wire object.

Throws IllegalArgumentException – If properties contains invalid wire types or case variants of the same key name.

16.15.8 public class WireAdminEvent

A Wire Admin Event.

WireAdminEvent objects are delivered to all registered WireAdminListener service objects which specify an interest in the WireAdminEvent type. Events must be delivered in chronological order with respect to each listener. For example, a WireAdminEvent of type WIRE_CONNECTED[p.34] must be delivered before a WireAdminEvent of type WIRE_DISCONNECTED[p.35] for a particular Wire object.

A type code is used to identify the type of event. The following event types are defined:

- WIRE_CREATED[p.35]
- WIRE_CONNECTED[p.34]
- WIRE_UPDATED[p.35]
- WIRE_TRACE[p.35]
- WIRE_DISCONNECTED[p.35]
- WIRE_DELETED[p.35]
- PRODUCER_EXCEPTION[p.34]
- CONSUMER_EXCEPTION[p.34]

Event type values must be unique and disjoint bit values. Event types must be defined as a bit in a 32 bit integer and can thus be bitwise OR'ed together.

Security Considerations. WireAdminEvent objects contain Wire objects. Care must be taken in the sharing of Wire objects with other bundles.

See Also WireAdminListener[p.36]

16.15.8.1 public static final int CONSUMER_EXCEPTION = 2

A Consumer service method has thrown an exception.

This WireAdminEvent type indicates that a Consumer service method has thrown an exception. The WireAdminEvent.getThrowable[p.36] method will return the exception that the Consumer service method raised.

The value of CONSUMER_EXCEPTION is 0x00000002.

16.15.8.2 public static final int PRODUCER_EXCEPTION = 1

A Producer service method has thrown an exception.

This WireAdminEvent type indicates that a Producer service method has thrown an exception. The WireAdminEvent.getThrowable[p.36] method will return the exception that the Producer service method raised.

The value of PRODUCER_EXCEPTION is 0x00000001.

16.15.8.3 public static final int WIRE_CONNECTED = 32

The WireAdminEvent type that indicates that an existing Wire object has become connected. The Consumer object and the Producer object that are associated with the Wire object have both been registered and the Wire object is connected. See Wire.isConnected[p.30] for a description of the connected state. This event may come before the producersConnected and consumersConnected method have returned or called to allow synchronous delivery of the events. Both methods can cause other WireAdminEvent s to take place and requiring this event to be send before these methods are returned would mandate asynchronous delivery.

The value of WIRE_CONNECTED is 0x00000020.

16.15.8.4 public static final int WIRE_CREATED = 4

A Wire has been created.

This WireAdminEvent type that indicates that a new Wire object has been created. An event is broadcast when WireAdmin.createWire[p.32] is called. The WireAdminEvent.getWire[p.36] method will return the Wire object that has just been created.

The value of WIRE_CREATED is 0x00000004.

16.15.8.5 public static final int WIRE_DELETED = 16

A Wire has been deleted.

This WireAdminEvent type that indicates that an existing wire has been deleted. An event is broadcast when WireAdmin.deleteWire[p.33] is called with a valid wire. WireAdminEvent.getWire[p.36] will return the Wire object that has just been deleted.

The value of WIRE_DELETED is 0x00000010.

16.15.8.6 public static final int WIRE_DISCONNECTED = 64

The WireAdminEvent type that indicates that an existing Wire object has become disconnected. The Consumer object or/and Producer object is/are unregistered breaking the connection between the two. See Wire.isConnected[p.30] for a description of the connected state.

The value of WIRE_DISCONNECTED is 0x00000040.

16.15.8.7 public static final int WIRE_TRACE = 128

The WireAdminEvent type that indicates that a new value is transferred over the Wire object. This event is sent after the Consumer service has been notified by calling the Consumer.updated[p.25] method or the Consumer service requested a new value with the Wire.poll[p.31] method. This is an advisory event meaning that when this event is received, another update may already have occurred and this the Wire.getLastValue[p.29] method returns a newer value then the value that was communicated for this event.

The value of WIRE_TRACE is 0x00000080.

16.15.8.8 public static final int WIRE_UPDATED = 8

A Wire has been updated.

This WireAdminEvent type that indicates that an existing Wire object has been updated with new properties. An event is broadcast when WireAdmin.updateWire[p.33] is called with a valid wire. The WireAdminEvent.getWire[p.36] method will return the Wire object that has just been updated.

The value of WIRE_UPDATED is 0x00000008.

16.15.8.9 public WireAdminEvent(ServiceReference reference, int type, Wire wire, Throwable exception)

reference The ServiceReference object of the Wire Admin service that created this event.

type The event type. See getType[p.36] .

wire The Wire object associated with this event.

exception An exception associated with this event. This may be null if no exception is associated with this event.

□ Constructs a WireAdminEvent object from the given ServiceReference object, event type, Wire object and exception.

16.15.8.10 public ServiceReference getServiceReference( )

□ Return the ServiceReference object of the Wire Admin service that created this event.

Returns The ServiceReference object for the Wire Admin service that created this event.

16.15.8.11 public Throwable getThrowable( )

□ Returns the exception associated with the event, if any.

Returns An exception or null if no exception is associated with this event.

16.15.8.12 public int getType( )

□ Return the type of this event.

The type values are:

- WIRE_CREATED[p.35]
- WIRE_CONNECTED[p.34]
- WIRE_UPDATED[p.35]
- WIRE_TRACE[p.35]
- WIRE_DISCONNECTED[p.35]
- WIRE_DELETED[p.35]
- PRODUCER_EXCEPTION[p.34]
- CONSUMER_EXCEPTION[p.34]

Returns The type of this event.

16.15.8.13 public Wire getWire( )

□ Return the Wire object associated with this event.

Returns The Wire object associated with this event or null when no Wire object is associated with the event.

16.15.9 public interface WireAdminListener

Listener for Wire Admin Events.

WireAdminListener objects are registered with the Framework service registry and are notified with a WireAdminEvent object when an event is broadcast.

WireAdminListener objects can inspect the received WireAdminEvent object to determine its type, the Wire object with which it is associated, and the Wire Admin service that broadcasts the event.

WireAdminListener objects must be registered with a service property WireConstants.WIREADMIN_EVENTS[p.38] whose value is a bitwise OR of all the event types the listener is interested in receiving.

For example:

```
 Integer mask = new Integer(WIRE_TRACE | WIRE_CONNECTED |
WIRE_DISCONNECTED);
 Hashtable ht = new Hashtable();
 ht.put(WIREADMIN_EVENTS, mask);
 context.registerService(WireAdminListener.class.getName(),
this, ht);
```

If a WireAdminListener object is registered without a service property WireConstants.WIREADMIN_EVENTS[p.38] , then the WireAdminListener will receive no events.

Security Considerations. Bundles wishing to monitor WireAdminEvent objects will require ServicePermission[REGISTER,WireAdminListener] to register a WireAdminListener service. Since WireAdminEvent objects contain Wire objects, care must be taken in assigning permission to register a WireAdminListener service.

See Also WireAdminEvent[p.33]

16.15.9.1 public void wireAdminEvent(WireAdminEvent event)

event The WireAdminEvent object.

□ Receives notification of a broadcast WireAdminEvent object. The event object will be of an event type specified in this WireAdminListener service's WireConstants.WIREADMIN_EVENTS[p.38] service property.

16.15.10 public interface WireConstants

Defines standard names for Wire properties, wire filter attributes, Consumer and Producer service properties.

16.15.10.1 public static final String WIREADMIN_CONSUMER_COMPOSITE = "wireadmin.consumer.composite"

A service registration property for a Consumer service that is composite. It contains the names of the composite Producer services it can cooperate with. Inter-operability exists when any name in this array matches any name in the array set by the Producer service. The type of this property must be String[].

16.15.10.2 public static final String WIREADMIN_CONSUMER_FLAVORS = "wireadmin.consumer.flavors"

Service Registration property (named wireadmin. consumer. flavors) specifying the list of data types understood by this Consumer service.

The Consumer service object must be registered with this service property. The list must be in the order of preference with the first type being the most preferred. The value of the property must be of type `Class[]`.

16.15.10.3 public static final String WIREADMIN_CONSUMER_PID = "wireadmin.consumer.pid"

`Wire` property key (named `wireadmin.consumer.pid`) specifying the `service.pid` of the associated Consumer service.

This wire property is automatically set by the Wire Admin service. The value of the property must be of type `String`.

16.15.10.4 public static final String WIREADMIN_CONSUMER_SCOPE = "wireadmin.consumer.scope"

Service registration property key (named `wireadmin.consumer.scope`) specifying a list of names that may be used to define the scope of this `Wire` object. A `Consumer` service should set this service property when it can produce more than one kind of value. This property is only used during registration, modifying the property must not have any effect of the `Wire` object's scope. Each name in the given list mist have `WirePermission[CONSUME]` or else is ignored. The type of this service registration property must be `String[]`.

See Also `Wire.getScope`[p.29], `WIREADMIN_PRODUCER_SCOPE`[p.40]

16.15.10.5 public static final String WIREADMIN_EVENTS = "wireadmin.events"

Service Registration property (named `wireadmin.events`) specifying the `WireAdminEvent` type of interest to a Wire Admin Listener service. The value of the property is a bitwise OR of all the `WireAdminEvent` types the Wire Admin Listener service wishes to receive and must be of type `Integer`.

See Also `WireAdminEvent`[p.33]

16.15.10.6 public static final String WIREADMIN_FILTER = "wireadmin.filter"

`Wire` property key (named `wireadmin.filter`) specifying a filter used to control the delivery rate of data between the Producer and the Consumer service.

This property should contain a filter as described in the `Filter` class. The filter can be used to specify when an updated value from the Producer service should be delivered to the Consumer service. In many cases the Consumer service does not need to receive the data with the same rate that the Producer service can generate data. This property can be used to control the delivery rate.

The filter can use a number of pre-defined attributes that can be used to control the delivery of new data values. If the filter produces a match upon the wire filter attributes, the Consumer service should be notifed of the updated data value.

If the Producer service was registered with the WIREADMIN_PRODUCER_FILTERS[p.39] service property indicating that the Producer service will perform the data filtering then the Wire object will not perform data filtering. Otherwise, the Wire object must perform basic filtering. Basic filtering includes supporting the following standard wire filter attributes:

- WIREVALUE_CURRENT[p.40] - Current value
- WIREVALUE_PREVIOUS[p.40] - Previous value
- WIREVALUE_DELTA_ABSOLUTE[p.40] - Absolute delta
- WIREVALUE_DELTA_RELATIVE[p.40] - Relative delta
- WIREVALUE_ELAPSED[p.40] - Elapsed time

See Also org.osgi.framework.Filter

16.15.10.7 public static final String WIREADMIN_PID = "wireadmin.pid"

Wire property key (named wireadmin.pid) specifying the persistent identity (PID) of this Wire object.

Each Wire object has a PID to allow unique and persistent identification of a specific Wire object. The PID must be generated by the WireAdmin[p.32] service when the Wire object is created.

This wire property is automatically set by the Wire Admin service. The value of the property must be of type String.

16.15.10.8 public static final String WIREADMIN_PRODUCER_COMPOSITE = "wireadmin.producer.composite"

A service registration property for a Producer service that is composite. It contains the names of the composite Consumer services it can inter-operate with. Inter-operability exists when any name in this array matches any name in the array set by the Consumer service. The type of this property must be String[].

16.15.10.9 public static final String WIREADMIN_PRODUCER_FILTERS = "wireadmin.producer.filters"

Service Registration property (named wireadmin.producer.filters). A Producer service registered with this property indicates to the Wire Admin service that the Producer service implements at least the filtering as described for the WIREADMIN_FILTER[p.38] property. If the Producer service is not registered with this property, the Wire object must perform the basic filtering as described in WIREADMIN_FILTER[p.38].

The type of the property value is not relevant. Only its presence is relevant.

16.15.10.10 public static final String WIREADMIN_PRODUCER_FLAVORS = "wireadmin.producer.flavors"

Service Registration property (named wireadmin.producer.flavors) specifying the list of data types available from this Producer service.

The Producer service object should be registered with this service property.

The value of the property must be of type Class[].

16.15.10.11 public static final String WIREADMIN_PRODUCER_PID =

"wireadmin.producer.pid"

`Wire` property key (named `wireadmin.producer.pid`) specifying the `service.pid` of the associated Producer service.

This wire property is automatically set by the WireAdmin service. The value of the property must be of type `String`.

16.15.10.12 public static final String WIREADMIN_PRODUCER_SCOPE = "wireadmin.producer.scope"

Service registration property key (named `wireadmin.producer.scope`) specifying a list of names that may be used to define the scope of this `Wire` object. A Producer service should set this service property when it can produce more than one kind of value. This property is only used during registration, modifying the property must not have any effect of the `Wire` object's scope. Each name in the given list mist have `WirePermission[PRODUCE, name]` or else is ignored. The type of this service registration property must be `String[]`.

See Also `Wire.getScope`[p.29], `WIREADMIN_CONSUMER_SCOPE`[p.38]

16.15.10.13 public static final String WIREADMIN_SCOPE_ALL

Matches all scope names.

16.15.10.14 public static final String WIREVALUE_CURRENT = "wirevalue.current"

`Wire` object's filter attribute (named `wirevalue.current`) representing the current value.

16.15.10.15 public static final String WIREVALUE_DELTA_ABSOLUTE = "wirevalue.delta.absolute"

`Wire` object's filter attribute (named `wirevalue.delta.absolute`) representing the absolute delta. The absolute (always positive) difference between the last update and the current value (only when numeric). This attribute must not be used when the values are not numeric.

16.15.10.16 public static final String WIREVALUE_DELTA_RELATIVE = "wirevalue.delta.relative"

`Wire` object's filter attribute (named `wirevalue.delta.relative`) representing the relative delta. The relative difference is |`previous`-`current` |/| `current`| (only when numeric). This attribute must not be used when the values are not numeric.

16.15.10.17 public static final String WIREVALUE_ELAPSED = "wirevalue.elapsed"

`Wire` object's filter attribute (named `wirevalue.elapsed`) representing the elapsed time, in ms, between this filter evaluation and the last update of the `Consumer` service.

16.15.10.18 public static final String WIREVALUE_PREVIOUS = "wirevalue.previous"

`Wire` object's filter attribute (named `wirevalue.previous`) representing the previous value.

16.15.11 public final class WirePermission extends BasicPermission

Permission for the scope of a `Wire` object. When a `Envelope` object is used for communication with the `poll` or `update` method, and the scope is set, then the `Wire` object must verify that the Consumer service has `WirePermission[name,CONSUME]` and the Producer service has `WirePermission[name,PRODUCE]` for all names in the scope.

The names are compared with the normal rules for permission names. This means that they may end with a "*" to indicate wildcards. E.g. Door.* indicates all scope names starting with the string "Door". The last period is required due to the implementations of the `BasicPermission` class.

16.15.11.1 public static final String CONSUME = "consume"

The action string for the `CONSUME` action: value is "consume".

16.15.11.2 public static final String PRODUCE = "produce"

The action string for the `PRODUCE` action: value is "produce".

16.15.11.3 public WirePermission(String name, String actions)

□ Create a new WirePermission with the given name (may be wildcard) and actions.

16.15.11.4 public boolean equals(Object obj)

obj The object to test for equality.

□ Determines the equalty of two `WirePermission` objects. Checks that specified object has the same name and actions as this `WirePermission` object.

Returns true if `obj` is a `WirePermission`, and has the same name and actions as this `WirePermission` object; `false` otherwise.

16.15.11.5 public String getActions( )

□ Returns the canonical string representation of the actions. Always returns present actions in the following order: `produce`, `consume`.

Returns The canonical string representation of the actions.

16.15.11.6 public int hashCode( )

□ Returns the hash code value for this object.

Returns Hash code value for this object.

16.15.11.7 public boolean implies(Permission p)

p The permission to check against.

□ Checks if this `WirePermission` object `implies` the specified permission.

More specifically, this method returns `true` if:

- *p* is an instanceof the `WirePermission` class,
- *p*'s actions are a proper subset of this object's actions, and
- *p*'s name is implied by this object's name. For example, `java.*` implies `java.home`.

Returns `true` if the specified permission is implied by this object; `false` otherwise.

16.15.11.8 public PermissionCollection newPermissionCollection( )

- Returns a new `PermissionCollection` object for storing `WirePermission` objects.

Returns A new `PermissionCollection` object suitable for storing `WirePermission` objects.

16.15.11.9 public String toString( )

- Returns a string describing this `WirePermission`. The convention is to specify the class name, the permission name, and the actions in the following format: '(org.osgi.service.wireadmin.WirePermission "name" "actions")'.

Returns information about this `Permission` object.

16.16 References

[48] *Design Patterns*
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison Wesley, ISBN 0-201-63361

21 Position Specification

Version 1.0

21.1 Introduction

The Position class is a utility providing bundle developers with a consistent way of handling geographic positions in OSGi applications. The Position class is intended to be used with the Wire Admin service but has wider applicability.

The Position class is designed to be compatible with the Global Positioning System (GPS). This specification will not define or explain the complexities of positioning information. It is assumed that the reader has the appropriate background to understand this information.

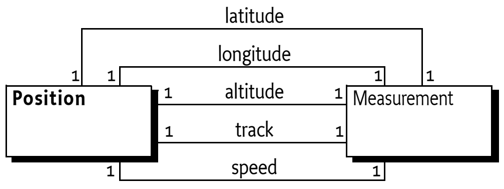
21.1.1 Essentials

- *Position* – Provide an information object that has well defined semantics for a position.
- *WGS-84* – Use the World Geodetic System 84 as the datum.
- *Speed* – Provide speed and track information.
- *Errors* – Position information always has certain errors or cannot be measured at all. This information must be available to the users of the information.
- *Units* – Use SI units for all measurements.
- *Wire Admin* – This specification must work within the Wire Admin service.

21.1.2 Entities

- *Position* – An object containing the different aspects of a position.
- *Measurement* – Contains a typed measurement made at a certain time and with a specified error.

Figure 67 Class Diagram, org.osgi.util.position



21.2 Positioning

The Position class is used to give information about the position and movement of a vehicle with a specified amount of uncertainty. The position is based on WGS-84.

The Position class offers the following information:

- getLatitude() – The WGS-84 latitude of the current position. The unit of a latitude must be rad (radians).
- getLongitude() – The WGS-84 longitude of the current position. The unit of a longitude must be rad (radians).
- getAltitude() – Altitude is expressed as height in meters above the WGS-84 ellipsoid. This value can differ from the actual height above mean sea level depending on the place on earth where the measurement is taken place. This value is not corrected for the geoid.
- getTrack() – The true north course of the vehicle in radians.
- getSpeed() – The ground speed. This speed must not include vertical speed.

21.3 Units

Longitude and latitude are represented in radians, not degrees. This is consistent with the use of the Measurement object. Radians can be converted to degrees with the following formula, when lonlat is the longitude or latitude:

```
degrees = (lonlat / π) * 180
```

Calculation errors are significantly reduced when all calculations are done with a single unit system. This approach increases the complexity of presentation, but presentations are usually localized and require conversion anyway. Also, the radians are the units in the SI system and the java.lang.Math class uses only radians for angles.

21.4 Optimizations

A Position object must be immutable. It must remain its original values after it is created.

The Position class is not final. This approach implies that developers are allowed to sub-class it and provide optimized implementations. For example, it is possible that the Measurement objects are only constructed when actually requested.

21.5 Errors

Positioning information is never exact. Even large errors can exist in certain conditions. For this reason, the Position class returns all its measurements as Measurement objects. The Measurement class maintains an error value for each measurement.

In certain cases it is not possible to supply a value; in those cases, the method should return a NaN as specified in the Measurement class.

21.6 Using Position With Wire Admin

The primary reason the Position is specified, is to use it with the *Wire Admin Service Specification* on page 1. A bundle that needs position information should register a Consumer service and the configuration should connect this service to an appropriate Producer service.

21.7 Related Standards

21.7.1 JSR 179

In JCP, started [65] *Location API for J2ME* . This API is targeted at embedded systems and is likely to not contain some of the features found in this API. This API is targeted to be reviewed at Q4 of 2002. This API should be considered in a following release.

21.8 Security

The security aspects of the Position class are delegated to the security aspects of the Wire Admin service. The Position object only carries the information. The Wire Admin service will define what Consumer services will receive position information from what Producer services. It is therefore up to the administrator of the Wire Admin service to assure that only trusted bundles receive this information, or can supply it.

21.9 org.osgi.util.position

The OSGi Position Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.util.position; specification-ver-
sion=1.0
```

21.9.1 public class Position

Position represents a geographic location, based on the WGS84 System (World Geodetic System 1984).

The org.osgi.util.measurement.Measurement class is used to represent the values that make up a position.

A given position object may lack any of it's components, i.e. the altitude may not be known. Such missing values will be represented by null.

Position does not override the implementation of either equals() or hash-Code() because it is not clear how missing values should be handled. It is up to the user of a position to determine how best to compare two position objects. A `Position` object is immutable.

21.9.1.1 public Position(Measurement lat, Measurement lon, Measurement alt, Measurement speed, Measurement track)

lat a `Measurement` object specifying the latitude in radians, or null

lon a `Measurement` object specifying the longitude in radians, or null

alt a `Measurement` object specifying the altitude in meters, or null

speed a `Measurement` object specifying the speed in meters per second, or null

track a `Measurement` object specifying the track in radians, or null

□ Contructs a `Position` object with the given values.

21.9.1.2 public Measurement getAltitude( )

□ Returns the altitude of this position in meters.

Returns a `Measurement` object in `Unit.m` representing the altitude in meters above the ellipsoid `null` if the altitude is not known.

21.9.1.3 public Measurement getLatitude( )

□ Returns the latitude of this position in radians.

Returns a `Measurement` object in `Unit.rad` representing the latitude, or `null` if the latitude is not known..

21.9.1.4 public Measurement getLongitude( )

□ Returns the longitude of this position in radians.

Returns a `Measurement` object in `Unit.rad` representing the longitude, or `null` if the longitude is not known.

21.9.1.5 public Measurement getSpeed( )

□ Returns the ground speed of this position in meters per second.

Returns a `Measurement` object in `Unit.m_s` representing the speed, or `null` if the speed is not known..

21.9.1.6 public Measurement getTrack( )

□ Returns the track of this position in radians as a compass heading. The track is the extrapolation of previous previously measured positions to a future position.

Returns a `Measurement` object in `Unit.rad` representing the track, or `null` if the track is not known..

21.10 References

[63] *World Geodetic System 84 (WGS-84)*
http://www.wgs84.com

[64] *Location Interoperability Forum*
http://www.locationforum.org/

[65] *Location API for J2ME*
http://www.jcp.org/jsr/detail/179.jsp

20 Measurement and State Specification

Version 1.0

20.1 Introduction

The Measurement class is a utility that provides a consistent way of handling a diverse range of measurements for bundle developers. Its purpose is to simplify the correct handling of measurements in OSGi Service Platforms.

OSGi bundle developers from all over the world have different preferences for measurement units, such as feet versus meters. In an OSGi environment, bundles developed in different parts of the world can and will exchange measurements when collaborating.

Distributing a measurement such as a simple floating point number requires the correct and equal understanding of the measurement's semantic by both the sender and the receiver. Numerous accidents have occurred due to misunderstandings between the sender and receiver because there are so many different ways to represent the same value. For example, on September 23, 1999, the Mars Polar Lander was lost because calculations used to program the craft's trajectory were input with English units while the operation documents specified metric units. See [62] *Mars Polar Lander failure* for more information.

This Measurement and State Specification defines the norm that should be used by all applications that execute in an OSGi Service Platform. This specification also provides utility classes.

20.1.1 Measurement Essentials

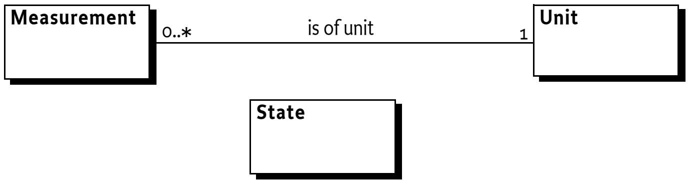
- *Numerical error* – All floating point measurements should be able to have a numerical error.
- *Numerical error calculations simplification* – Support should be provided to simplify measurements calculations.
- *Unit conflict resolution* – It must not be possible to perform addition or subtraction with different units when they are not compatible. For example, it must not be possible to add meters to amperes or watts to pascals.
- *Unit coercion* – Multiplication and division operations involving more than one type of measurement must result in a different unit. For example, if meters are divided by seconds, the result must be a new unit that represents m/s.
- *Time-stamp* – Measurements should contain a time-stamp so that bundles can determine the age of a particular measurement.

- *Support for floating and discrete values* – Both floating point values (64 bit Java double floats) and discrete measurements (32 bit Java int) should be supported.
- *Consistency* – The method of error calculation and handling of unit types should be consistent.
- *Presentation* – The format of measurements and specified units should be easy to read and understand.

20.1.2 Measurement Entities

- *Measurement object* – A Measurement object contains a double value, a double error, and a long time-stamp. It is associated with a Unit object that represents its *type*.
- *State object* – A State object contains a discrete measurement (int) with a time-stamp and a name.
- *Unit object* – A Unit object represents a unit such as meter, second, mol, or Pascal. A number of Unit objects are predefined and have common names. Other Unit objects are created as needed from the 7 basic Système International d'Unité (SI) units. Different units are *not* used when a conversion is sufficient. For example, the unit of a Measurement object for length is *always* meters. If the length is needed in feet, then the number of feet is calculated by multiplying the value of the Measurement object in meters with the necessary conversion factor.
- *Error* – When a measurement is taken, it is *never* accurate. This specification defines the error as the value that is added and subtracted to the value to produce an interval, where the probability is 95% that the actual value falls within this interval.
- *Unit* – A unit is the *type* of a measurement: meter, feet, liter, gallon etc.
- *Base Unit* – One of the 7 base units defined in the SI.
- *Derived SI unit* – A unit is a derived SI unit when it is a combination of exponentiated base units. For example, a volt (V) is a derived unit because it can be expressed as $(m^2 \times kg) / (s^3 \times A)$, where m, kg, s and A are all base units.
- *Quantitative derivation* – A unit is quantitatively derived when it is converted to one of the base units or derived units using a conversion formula. For example, kilometers (km) can be converted to meters (m), gallons can be converted to liters, or horsepower can be converted to watts.

Figure 64 *Class Diagram, org.osgi.util.measurement*



20.2 Measurement Object

A Measurement object contains a value, an error, and a time-stamp It is linked to a Unit object that describes the measurement unit in an SI Base Unit or Derived SI Unit.

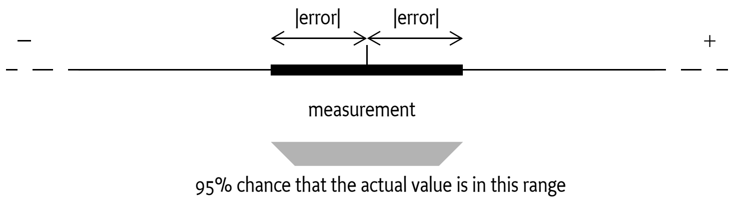
20.2.1 Value

The value of the Measurement object is the measured value. It is set in a constructor. The type of the value is double.

20.2.2 Error

The Measurement object can contain a numerical error. This error specifies an interval by adding and subtracting the error value from the measured value. The type of the error is double. A valid error value indicates that the actual measured value has a 95% chance of falling within this interval (see Figure 2). If the error is not known it should be represented as a Double.NaN.

Figure 65 *The Error Interval*



20.2.3 Time-stamp

When a Measurement object is created, the time-stamp can be set. A time-stamp is a long value representing the number of milliseconds since the epoch midnight of January 1, 1970, UTC (this is the value from System.currentTimeMillis() method).

By default, a time-stamp is not set because the call to System.currentTimeMillis() incurs overhead. If the time-stamp is not set when the Measurement object is created, then its default value is zero. If the time-stamp is set, the creator of the Measurement object must give the time as an argument to the constructor. For example:

```
Measurement m = new Measurement(
   v, e, null, System.currentTimeMillis() );
```

20.3 Error Calculations

Once a measurement is taken, it often is used in calculations. The error value assigned to the result of a calculation depends largely on the error values of the operands. Therefore, the Measurement class offers addition, subtraction, multiplication, and division functions for measurements and constants. These functions take the error into account when performing the specific operation.

The Measurement class uses absolute errors and has methods to calculate a new absolute error when multiplication, division, addition, or subtraction is performed. Error calculations must therefore adhere to the rules listed in Table 25. In this table, Δa is the absolute positive error in a value a and Δb is the absolute positive error in a value b. c is a constant floating point value without an error.

Calculation	Function	Error
$a \times b$	mul(Measurement)	$\| \Delta a \times b \| + \| a \times \Delta b \|$
a / b	div(Measurement)	$(\| \Delta a \times b \| + \| a \times \Delta b \|) / b^2$
$a + b$	add(Measurement)	$\Delta a + \Delta b$
$a - b$	sub(Measurement)	$\Delta a + \Delta b$
$a \times c$	mul(double)	$\| \Delta a \times c \|$
a / c	div(double)	$\| \Delta a / c \|$
$a + c$	add(double)	Δa
$a - c$	sub(double)	Δa

Table 25 Error Calculation Rules

20.4 Comparing Measurements

Measurement objects have a value and an error range, making comparing these objects more complicated than normal scalars.

20.4.1 Identity and Equality

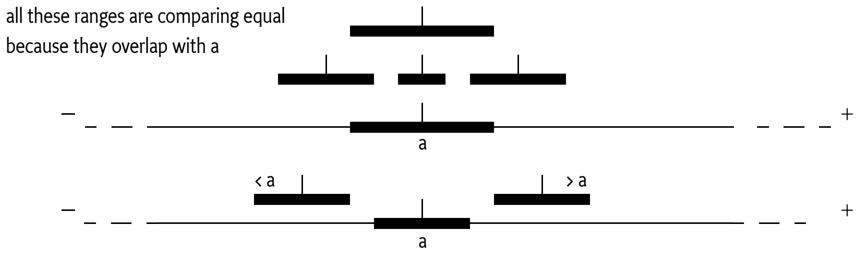
Both equals(Object) and hashCode() methods are overridden to provide value-based equality. Two Measurement objects are equal when the unit, error, and value are the same. The time-stamp is not relevant for equality or the hash code.

20.4.2 Comparing Measurement Objects

The Measurement class implements the java.lang.Comparable interface and thus implements the compareTo(Object) method. Comparing two Measurement objects is not straightforward, however, due to the associated error. The error effectively creates a range, so comparing two Measurement objects is actually comparing intervals.

Two Measurement objects are considered to be equal when their intervals overlap. In all other cases, the value is used in the comparison.

Figure 66 *Comparing Measurement Objects*


all these ranges are comparing equal
because they overlap with a
–
+
a
< a
> a
–
+
a

This comparison implies that the equals(Object) method may return false while the compareTo(Object) method returns 0 for the same Measurement object.

20.5 Unit Object

Each Measurement object is related to a Unit object. The Unit object defines the unit of the measurement value and error. For example, the Unit object might define the unit of the measurement value and the error as meters (m). For convenience, the Unit class defines a number of standard units as constants. Measurement objects are given a specific Unit with the constructor. The following example shows how a measurement can be associated with meters (m):

```
Measurement length = new Measurement( v, 0.01, Unit.m );
```

Units are based on the Système International d'Unité (SI), developed after the French Revolution. The SI consists of 7 different units that can be combined in many ways to form a large series of derived units. The basic 7 units are listed in Table 26. For more information, see [58] *General SI index*.

Description	Unit name	Symbol
length	meter	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	mole	mol
luminous intensity	candela	cd

Table 26 *Basic SI units.*

Additional units are derived in the following ways:

Derived units can be a combination of exponentiated base units. For example, Hz (Hertz) is the unit for frequencies and is actually derived from the calculation of 1/s. A more complicated derived unit is volt (V). A volt is actually:

$$(m^2 \times kg) / (s^3 \times A)$$

The SI defines various derived units with their own name, for example pascal (Pa), watt (W), volt (V), and many more.

The Measurement class must maintain its unit by keeping track of the exponents of the 7 basic SI units.

If different units are used in addition or subtraction of Measurement objects, an ArithmeticException must be thrown.

```
Measurement length = new Measurement( v1, 0.01, Unit.m );
Measurement duration = new Measurement( v2, 0, Unit.s );
try {
  Measurement r = length.add( duration );
}
catch( ArithmeticException e ) {
  // This must be thrown
}
```

When two Measurement objects are multiplied, the Unit object of the result contains the sum of the exponents. When two Measurement objects are divided, the exponents of the Unit object of the result are calculated by subtraction of the exponents.

The Measurement class must support exponents of -64 to +63. Overflow must not be reported but must result in an invalid Unit object. All calculations with an invalid Unit object should result in an invalid Unit object. Typical computations generate exponents for units between +/- 4.

20.5.1 Quantitive Differences

The base and derived units can be converted to other units that are of the same *quality*, but require a conversion because their scales and offsets may differ. For example, degrees Fahrenheit, kelvin, and Celsius are all temperatures and, therefore, only differ in their quantity. Kelvin and Celsius are the same scale and differ only in their starting points. Fahrenheit differs from kelvin in that both scale and starting point differ.

Using different Unit objects for the units that differ only in quantity can easily introduce serious software bugs. Therefore, the Unit class utilizes the SI units. Any exchange of measurements should be done using SI units to prevent these errors. When a measurement needs to be displayed, the presentation logic should perform the necessary conversions to present it in a localized form. For example, when speed is presented in a car purchased in the United States, it should be presented as miles instead of meters.

20.5.2 Why Use SI Units?

The adoption of the SI in the United States and the United Kingdom has met with resistance. This issue raises the question why the SI system has to be the preferred measurement system in the OSGi Specifications.

The SI system is utilized because it is the only measurement *system* that has a consistent set of base units. The base units can be combined to create a large number of derived units without requiring a large number of complicated conversion formulas. For example, a watt is simply a combination of meters, kilograms, and seconds ($m^2 \times kg/s^3$). In contrast, horsepower is not easily related to inches, feet, fathoms, yards, furlongs, ounces, pounds, stones, or miles. This difficulty is the reason that science has utilized the SI for a long time. It is also the reason that the SI has been chosen as the system used for the `Measurement` class.

The purpose of the `Measurement` class is internal, however, and should not restrict the usability of the OSGi environment. Users should be able to use the local measurement units when data is input or displayed. This choice is the responsibility of the application developer.

20.6 State Object

The `State` object is used to represent discrete states. It contains a time-stamp but does not contain an error or `Unit` object. The `Measurement` object is not suitable to maintain discrete states. For example, a car door can be `LOCKED`, `UNLOCKED`, or `CHILDLOCKED`. Measuring and operating with these values does not require error calculations, nor does it require SI units. Therefore, the `State` object is a simple, named object that holds an integer value.

20.7 Related Standards

20.7.1 JSR 108 Units Specification

Sun Microsystems Java Community Process (JCP) [59] *JSR 108 Units Specification* addresses the same issues as this Measurement and State Specification. At the time of the writing of this specification, no public review of the JCP has occurred. This JSR, however, seems to be based on the [60] *Unidata user group: MetaApps project.* This Unidata API overlaps this specification but has the following issues:

- It uses a significant number of classes. Using many small classes can create significant overhead, which is a problem for the resource-constrained OSGi Service Platform.
- It treats the SI units in the same way as quantitatively derived units. As explained earlier, the purpose of the `Measurement` class is to prevent confusion between units that only differ in their quantity like gallons and liters. It is considered better to strictly separate the `presentation` of units from the units used in calculations.
- It is not yet complete as of the writing of this specification.

This JSR does not seem to move past the initial review phase.

20.7.2 GNU Math Library in Kawa

The open source project Kawa, a scheme-based Java environment, has included a gnu.math library that contains unit handling similar to this specification. It can be found at [61] *A Math Library containing unit handling in Kawa.*

The library seems considerably more complex without offering much more functionality than this specification. It also does not strictly separate basic SI units such as meter from quantitatively derived units such as pica.

20.8 Security Considerations

The Measurement, Unit, and State classes have been made immutable. Instances of these classes can be freely handed out to other bundles because they cannot be extended, nor can the value, error, or time-stamp be altered after the object is created.

20.9 org.osgi.util.measurement

Unexpected tag, assume para (x=49,y=2) [p.9]

Create a new <tt>Measurement</tt> object from a String. The format of the string must be: <value>':' <unit>[':' error] whitespace is not allowed

Unexpected tag, assume para (x=69,y=2) [p.9]

Create a new <tt>Measurement</tt> object from a String. The format of the string must be: <value>':' <unit>[':' error] whitespace is not allowed

The OSGi Measurement Package. Specification Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.util.measurement; specification-ver-
sion=1.1
```

20.9.1 Summary

- Measurement - Represents a value with an error, a unit and a time-stamp. [p.8]
- State - Groups a state name, value and timestamp. [p.13]
- Unit - A unit system for measurements. [p.3]

20.9.2 public class Measurement implements Comparable

Represents a value with an error, a unit and a time-stamp.

A Measurement object is used for maintaining the tuple of value, error, unit and time-stamp. The value and error are represented as doubles and the time is measured in milliseconds since midnight, January 1, 1970 UTC.

Mathematic methods are provided that correctly calculate taking the error into account. A runtime error will occur when two measurements are used in an incompatible way. E.g., when a speed (m/s) is added to a distance (m). The measurement class will correctly track changes in unit during multiplication and division, always coercing the result to the most simple form. See `Unit`[p.3] for more information on the supported units.

Errors in the measurement class are absolute errors. Measurement errors should use the P95 rule. Actual values must fall in the range value +/- error 95% or more of the time.

A `Measurement` object is immutable in order to be easily shared.

Note: This class has a natural ordering that is inconsistent with equals. See `compareTo`[p.10] .

20.9.2.1 public Measurement(String encoded)

encoded The encoded value of the `Measurement`.

□ Create a new `Measurement` object from a String. The format of the string must be: ‘:’ [‘:’ error] whitespace is not allowed

Throws `IllegalArgumentException` – If the encoded value cannot be properly parsed.

20.9.2.2 public Measurement(double value, double error, Unit unit, long time)

value The value of the `Measurement`.

error The error of the `Measurement`.

unit The `Unit` object in which the value is measured. If this argument is `null`, then the unit will be set to `Unit.unity`[p.16] .

time The time measured in milliseconds since midnight, January 1, 1970 UTC.

□ Create a new `Measurement` object.

20.9.2.3 public Measurement(double value, double error, Unit unit)

value The value of the `Measurement`.

error The error of the `Measurement`.

unit The `Unit` object in which the value is measured. If this argument is `null`, then the unit will be set to `Unit.unity`[p.16] .

□ Create a new `Measurement` object with a time of zero.

20.9.2.4 public Measurement(double value, Unit unit)

value The value of the `Measurement`.

unit The `Unit` in which the value is measured. If this argument is `null`, then the unit will be set to `Unit.unity`[p.16] .

□ Create a new `Measurement` object with an error of 0.0 and a time of zero.

20.9.2.5 public Measurement(double value)

value The value of the `Measurement`.

□ Create a new `Measurement` object with an error of 0.0, a unit of `Unit.unity`[p.16] and a time of zero.

20.9.2.6 public Measurement add(Measurement m)

m The `Measurement` object that will be added with this object.

□ Returns a new `Measurement` object that is the sum of this object added to the specified object. The error and unit of the new object are computed. The time of the new object is set to the time of this object.

Returns A new `Measurement` object that is the sum of this and m.

Throws `ArithmeticException` – If the `Unit` objects of this object and the specified object cannot be added.

See Also `Unit`[p.3]

20.9.2.7 public Measurement add(double d, Unit u)

d The value that will be added with this object.

u The `Unit` object of the specified value.

□ Returns a new `Measurement` object that is the sum of this object added to the specified value.

Returns A new `Measurement` object that is the sum of this object added to the specified value. The unit of the new object is computed. The error and time of the new object is set to the error and time of this object.

Throws `ArithmeticException` – If the `Unit` objects of this object and the specified value cannot be added.

See Also `Unit`[p.3]

20.9.2.8 public Measurement add(double d)

d The value that will be added with this object.

□ Returns a new `Measurement` object that is the sum of this object added to the specified value.

Returns A new `Measurement` object that is the sum of this object added to the specified value. The error, unit, and time of the new object is set to the error, `Unit` and time of this object.

20.9.2.9 public int compareTo(Object obj)

obj The object to be compared.

□ Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.

Note: This class has a natural ordering that is inconsistent with equals. For this method, another `Measurement` object is considered equal if there is some `x` such that

```
getValue() - getError() <= x <= getValue() + getError()
```

for both `Measurement` objects being compared.

Returns A negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.

Throws `ClassCastException` – If the specified object is not of type `Measurement`.

`ArithmeticException` – If the unit of the specified `Measurement` object is not equal to the `Unit` object of this object.

20.9.2.10 public Measurement div(Measurement m)

m The `Measurement` object that will be the divisor of this object.

☐ Returns a new `Measurement` object that is the quotient of this object divided by the specified object.

Returns A new `Measurement` object that is the quotient of this object divided by the specified object. The error and unit of the new object are computed. The time of the new object is set to the time of this object.

Throws `ArithmeticException` – If the `Unit` objects of this object and the specified object cannot be divided.

See Also `Unit`[p.3]

20.9.2.11 public Measurement div(double d, Unit u)

d The value that will be the divisor of this object.

u The `Unit` object of the specified value.

☐ Returns a new `Measurement` object that is the quotient of this object divided by the specified value.

Returns A new `Measurement` that is the quotient of this object divided by the specified value. The error and unit of the new object are computed. The time of the new object is set to the time of this object.

Throws `ArithmeticException` – If the `Unit` objects of this object and the specified object cannot be divided.

See Also `Unit`[p.3]

20.9.2.12 public Measurement div(double d)

d The value that will be the divisor of this object.

☐ Returns a new `Measurement` object that is the quotient of this object divided by the specified value.

Returns A new `Measurement` object that is the quotient of this object divided by the specified value. The error of the new object is computed. The unit and time of the new object is set to the `Unit` and time of this object.

20.9.2.13 public boolean equals(Object obj)

obj The object to compare with this object.

☐ Returns whether the specified object is equal to this object. Two `Measurement` objects are equal if they have same value, error and `Unit`.

Note: This class has a natural ordering that is inconsistent with equals. See `compareTo`[p.10] .

Returns `true` if this object is equal to the specified object; `false` otherwise.

20.9.2.14 public final double getError( )

☐ Returns the error of this `Measurement` object. The error is always a positive value.

Returns The error of this `Measurement` as a double.

20.9.2.15 public final long getTime( )

□ Returns the time at which this `Measurement` object was taken. The time is measured in milliseconds since midnight, January 1, 1970 UTC, or zero when not defined.

Returns The time at which this `Measurement` object was taken or zero.

20.9.2.16 public final Unit getUnit( )

□ Returns the `Unit` object of this `Measurement` object.

Returns The `Unit` object of this `Measurement` object.

See Also `Unit`[p.3]

20.9.2.17 public final double getValue( )

□ Returns the value of this `Measurement` object.

Returns The value of this `Measurement` object as a double.

20.9.2.18 public int hashCode( )

□ Returns a hash code value for this object.

Returns A hash code value for this object.

20.9.2.19 public Measurement mul(Measurement m)

m The `Measurement` object that will be multiplied with this object.

□ Returns a new `Measurement` object that is the product of this object multiplied by the specified object.

Returns A new `Measurement` that is the product of this object multiplied by the specified object. The error and unit of the new object are computed. The time of the new object is set to the time of this object.

Throws `ArithmeticException` – If the `Unit` objects of this object and the specified object cannot be multiplied.

See Also `Unit`[p.3]

20.9.2.20 public Measurement mul(double d, Unit u)

d The value that will be multiplied with this object.

u The `Unit` of the specified value.

□ Returns a new `Measurement` object that is the product of this object multiplied by the specified value.

Returns A new `Measurement` object that is the product of this object multiplied by the specified value. The error and unit of the new object are computed. The time of the new object is set to the time of this object.

Throws `ArithmeticException` – If the units of this object and the specified value cannot be multiplied.

See Also `Unit`[p.3]

20.9.2.21 public Measurement mul(double d)

d The value that will be multiplied with this object.

□ Returns a new `Measurement` object that is the product of this object multiplied by the specified value.

Returns A new `Measurement` object that is the product of this object multiplied by the specified value. The error of the new object is computed. The unit and time of the new object is set to the unit and time of this object.

20.9.2.22 public Measurement sub(Measurement m)

m The `Measurement` object that will be subtracted from this object.

□ Returns a new `Measurement` object that is the subtraction of the specified object from this object.

Returns A new `Measurement` object that is the subtraction of the specified object from this object. The error and unit of the new object are computed. The time of the new object is set to the time of this object.

Throws `ArithmeticException` – If the `Unit` objects of this object and the specified object cannot be subtracted.

See Also `Unit`[p.3]

20.9.2.23 public Measurement sub(double d, Unit u)

d The value that will be subtracted from this object.

u The `Unit` object of the specified value.

□ Returns a new `Measurement` object that is the subtraction of the specified value from this object.

Returns A new `Measurement` object that is the subtraction of the specified value from this object. The unit of the new object is computed. The error and time of the new object is set to the error and time of this object.

Throws `ArithmeticException` – If the `Unit` objects of this object and the specified object cannot be subtracted.

See Also `Unit`[p.3]

20.9.2.24 public Measurement sub(double d)

d The value that will be subtracted from this object.

□ Returns a new `Measurement` object that is the subtraction of the specified value from this object.

Returns A new `Measurement` object that is the subtraction of the specified value from this object. The error, unit and time of the new object is set to the error, `Unit` object and time of this object.

20.9.2.25 public String toString( )

□ Returns a `String` object representing this `Measurement` object.

Returns a `String` object representing this `Measurement` object.

20.9.3 public class State

Groups a state name, value and timestamp.

The state itself is represented as an integer and the time is measured in milliseconds since midnight, January 1, 1970 UTC.

A `State` object is immutable so that it may be easily shared.

20.9.3.1 public State(int value, String name, long time)

value The value of the state.

name The name of the state.

time The time measured in milliseconds since midnight, January 1, 1970 UTC.

□ Create a new `State` object.

20.9.3.2 public State(int value, String name)

value The value of the state.

name The name of the state.

□ Create a new `State` object with a time of 0.

20.9.3.3 public boolean equals(Object obj)

obj The object to compare with this object.

□ Return whether the specified object is equal to this object. Two `State` objects are equal if they have same value and name.

Returns `true` if this object is equal to the specified object; `false` otherwise.

20.9.3.4 public final String getName( )

□ Returns the name of this `State`.

Returns The name of this `State` object.

20.9.3.5 public final long getTime( )

□ Returns the time with which this `State` was created.

Returns The time with which this `State` was created. The time is measured in milliseconds since midnight, January 1, 1970 UTC.

20.9.3.6 public final int getValue( )

□ Returns the value of this `State`.

Returns The value of this `State` object.

20.9.3.7 public int hashCode( )

□ Returns a hash code value for this object.

Returns A hash code value for this object.

20.9.3.8 public String toString( )

□ Returns a `String` object representing this object.

Returns a `String` object representing this object.

20.9.4 public class Unit

A unit system for measurements. This class contains definitions of the most common SI units.

This class only support exponents for the base SI units in the range -64 to +63. Any operation which produces an exponent outside of this range will result in a `Unit` object with undefined exponents.

20.9.4.1 public static final Unit A

The electric current unit ampere (A)

20.9.4.2 public static final Unit C

The electric charge unit coulomb (C).

coulomb is expressed in SI units as s· A

20.9.4.3 public static final Unit cd

The luminous intensity unit candela (cd)

20.9.4.4 public static final Unit F

The capacitance unit farad (F).

farad is equal to C/V or is expressed in SI units as s 4 · A 2 /m 2 · kg

20.9.4.5 public static final Unit Gy

The absorbed dose unit gray (Gy).

Gy is equal to J/kg or is expressed in SI units as m 2 /s 2

20.9.4.6 public static final Unit Hz

The frequency unit hertz (Hz).

hertz is expressed in SI units as 1/s

20.9.4.7 public static final Unit J

The energy unit joule (J).

joule is equal to N· m or is expressed in SI units as m 2 · kg/s 2

20.9.4.8 public static final Unit K

The temperature unit kelvin (K)

20.9.4.9 public static final Unit kat

The catalytic activity unit katal (kat).

katal is expressed in SI units as mol/s

20.9.4.10 public static final Unit kg

The mass unit kilogram (kg)

20.9.4.11 public static final Unit lx

The illuminance unit lux (lx).

lux is expressed in SI units as cd/m 2

20.9.4.12 public static final Unit m

The length unit meter (m)

20.9.4.13 public static final Unit m2

The area unit square meter(m^2)

20.9.4.14 public static final Unit m3

The volume unit cubic meter (m^3)

20.9.4.15 public static final Unit m_s

The speed unit meter per second (m/s)

20.9.4.16 public static final Unit m_s2

The acceleration unit meter per second squared (m/s^2)

20.9.4.17 public static final Unit mol

The amount of substance unit mole (mol)

20.9.4.18 public static final Unit N

The force unit newton (N).

N is expressed in SI units as $m \cdot kg/s^2$

20.9.4.19 public static final Unit Ohm

The electric resistance unit ohm.

ohm is equal to V/A or is expressed in SI units as $m^2 \cdot kg/s^3 \cdot A^2$

20.9.4.20 public static final Unit Pa

The pressure unit pascal (Pa).

Pa is equal to N/m^2 or is expressed in SI units as $kg/m \cdot s^2$

20.9.4.21 public static final Unit rad

The angle unit radians (rad)

20.9.4.22 public static final Unit S

The electric conductance unit siemens (S).

siemens is equal to A/V or is expressed in SI units as $s^3 \cdot A^2/m^2 \cdot kg$

20.9.4.23 public static final Unit s

The time unit second (s)

20.9.4.24 public static final Unit T

The magnetic flux density unit tesla (T).

tesla is equal to Wb/m^2 or is expressed in SI units as $kg/s^2 \cdot A$

20.9.4.25 public static final Unit unity

No Unit (Unity)

20.9.4.26 public static final Unit V

The electric potential difference unit volt (V).

volt is equal to W/A or is expressed in SI units as m 2 · kg/s 3 · A

20.9.4.27 public static final Unit W

The power unit watt (W).

watt is equal to J/s or is expressed in SI units as m 2 · kg/s 3

20.9.4.28 public static final Unit Wb

The magnetic flux unit weber (Wb).

weber is equal to V· s or is expressed in SI units as m 2 · kg/s 2 · A

20.9.4.29 public boolean equals(Object obj)

obj the `Unit` object that should be checked for equality

□ Checks whether this `Unit` object is equal to the specified `Unit` object. The `Unit` objects are considered equal if their exponents are equal.

Returns true if the specified `Unit` object is equal to this `Unit` object.

20.9.4.30 public static Unit fromString(String unit)

20.9.4.31 public int hashCode( )

□ Returns the hash code for this object.

Returns This object's hash code.

20.9.4.32 public String toString( )

□ Returns a `String` object representing the `Unit`

Returns A `String` object representing the `Unit`

20.10 References

[57] *SI Units information*
http://physics.nist.gov/cuu/Units

[58] *General SI index*
http://directory.google.com/Top/Science/Reference/Units_of_Measurement

[59] *JSR 108 Units Specification*
http://www.jcp.org/jsr/detail/108.jsp

[60] *Unidata user group: MetaApps project*
http://www.unidata.ucar.edu/community/committees/metapps/docs/ucar/units/package-summary.html

[61] *A Math Library containing unit handling in Kawa*
http://www.gnu.org/software/kawa

[62] *Mars Polar Lander failure*
http://mars.jpl.nasa.gov/msp98/news/mc990930.html

9 Preferences Service Specification

Version 1.0

9.1 Introduction

Many bundles need to save some data persistently--in other words, the data is required to survive the stopping and restarting of the bundle, Framework and OSGi Service Platform. In some cases, the data is specific to a particular user. For example, imagine a bundle that implements some kind of game. User specific persistent data could include things like the user's preferred difficulty level for playing the game. Some data is not specific to a user, which we call *system* data. An example would be a table of high scores for the game.

Bundles which need to persist data in an OSGi environment can use the file system via org.osgi.framework.BundleContext.getDataFile. A file system, however, can store only bytes and characters, and provides no direct support for named values and different data types.

A popular class used to address this problem for Java applications is the java.util.Properties class. This class allows data to be stored as key/value pairs, called *properties*. For example, a property could have a name com.acme.fudd and a value of elmer. The Properties class has rudimentary support for storage and retrieving with its load and store methods. The Properties class, however, has the following limitations:

- Does not support a naming hierarchy.
- Only supports String property values.
- Does not allow its content to be easily stored in a back-end system.
- Has no user name-space management.

Since the Properties class was introduced in Java 1.0, efforts have been undertaken to replace it with a more sophisticated mechanism. One of these efforts is this Preferences Service specification.

9.1.1 Essentials

The focus of this specification is simplicity, not reliable access to stored data. This specification does *not* define a general database service with transactions and atomicity guarantees. Instead, it is optimized to deliver the stored information when needed, but it will return defaults, instead of throwing an exception, when the back-end store is not available. This approach may reduce the reliability of the data, but it makes the service easier to use, and allows for a variety of compact and efficient implementations.

This API is made easier to use by the fact that many bundles can be written to ignore any problems that the Preferences Service may have in accessing the back-end store, if there is one. These bundles will mostly or exclusively use the methods of the Preferences interface which are not declared to throw a BackingStoreException.

This service only supports the storage of scalar values and byte arrays. It is not intended for storing large data objects like documents or images. No standard limits are placed on the size of data objects which can be stored, but implementations are expected to be optimized for the handling of small objects.

A hierarchical naming model is supported, in contrast to the flat model of the Properties class. A hierarchical model maps naturally to many computing problems. For example, maintaining information about the positions of adjustable seats in a car requires information for each seat. In a hierarchy, this information can be modeled as a node per seat.

A potential benefit of the Preferences Service is that it allows user specific preferences data to be kept in a well defined place, so that a user management system could locate it. This benefit could be useful for such operations as cleaning up files when a user is removed from the system, or to allow a user's preferences to be cloned for a new user.

The Preferences Service does *not* provide a mechanism to allow one bundle to access the preferences data of another. If a bundle wishes to allow another bundle to access its preferences data, it can pass a Preferences or PreferencesService object to that bundle.

The Preferences Service is not intended to provide configuration management functionality. For information regarding Configuration Management, refer to the *Configuration Admin Service Specification* on page 33.

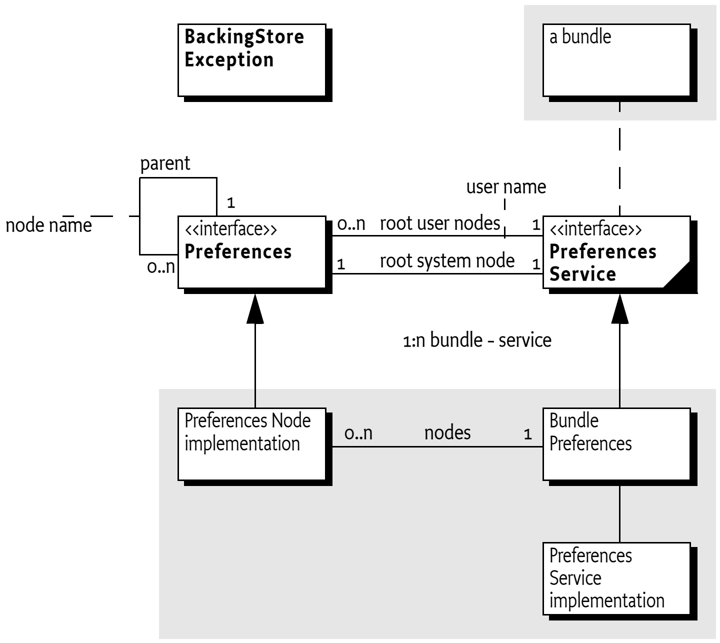
9.1.2 Entities

The PreferencesService is a relatively simple service. It provides access to the different roots of Preferences trees. A single system root node and any number of user root nodes are supported. Each *node* of such a tree is an object that implements the Preferences interface.

This Preferences interface provides methods for traversing the tree, as well as methods for accessing the properties of the node. This interface also contains the methods to flush data into persistent storage, and to synchronize the in-memory data cache with the persistent storage.

All nodes except root nodes have a parent. Nodes can have multiple children.

Figure 30 *Preferences Class Diagram*



9.1.3 Operation

The purpose of the Preferences Service specification is to allow bundles to store and retrieve properties stored in a tree of nodes, where each node implements the Preferences interface. The PreferencesService interface allows a bundle to create or obtain a Preferences tree for system properties, as well as a Preferences tree for each user of the bundle.

This specification allows for implementations where the data is stored locally on the service platform or remotely on a back-end system.

9.2 Preferences Interface

Preferences is an interface that defines the methods to manipulate a node and the tree to which it belongs. A Preferences object contains:

- A set of properties in the form of key/value pairs.
- A parent node.
- A number of child nodes.

9.2.1 Hierarchies

A valid Preferences object always belongs to a *tree*. A tree is identified by its root node. In such a tree, a Preferences object always has a single parent, except for a root node which has a null parent.

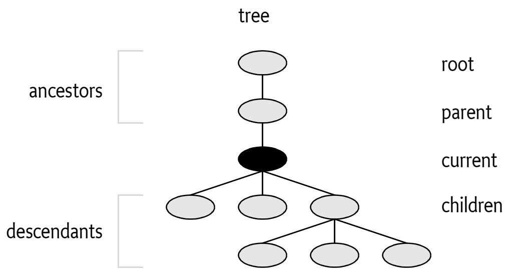
The root node of a tree can be found by recursively calling the parent() method of a node until null is returned. The nodes that are traversed this way are called the *ancestors* of a node.

Each Preferences object has a private name-space for child nodes. Each child node has a name that must be unique among its siblings. Child nodes are created by getting a child node with the `node(String)` method. The `String` argument of this call contains a path name. Path names are explained in the next section.

Child nodes can have child nodes recursively. These objects are called the *descendants* of a node.

Descendants are automatically created when they are obtained from a `Preferences` object, including any intermediate nodes that are necessary for the given path. If this automatic creation is not desired, the `nodeExists(String)` method can be used to determine if a node already exists.

Figure 31 *Categorization of nodes in a tree*



9.2.2 Naming

Each node has a name relative to its parent. A name may consist of Unicode characters except for the forward slash ("/"). There are no special names, like ".." or ".".

Empty names are reserved for root nodes. Node names that are directly created by a bundle must *always* contain at least one character.

Preferences node names and property keys are *case sensitive*: for example, "`org.osgi`" and "`oRg.oSgI`" are two distinct names.

The Preferences Service supports different roots, so there is no absolute root for the Preferences Service. This concept is similar to [21] *Windows Registry* that also supports a number of roots.

A path consists of one or more node names, separated by a slash ("/"). Paths beginning with a "/" are called *absolute paths* while other paths are called *relative paths*. Paths cannot end with a "/" except for the special case of the root node which has absolute path "/".

Path names are always associated with a specific node; this node is called the current node in the following descriptions. Paths identify nodes as follows.

- *Absolute path* – The first "/" is removed from the path, and the remainder of the path is interpreted as a relative path from the tree's root node.
- *Relative path* –
  - If the path is the empty string, it identifies the current node.
  - If the path is a name (does not contain a "/"), then it identifies the child node with that name.

- Otherwise, the first name from the path identifies a child of the current node. The name and slash are then removed from the path, and the remainder of the path is interpreted as a relative path from the child node.

9.2.3 Tree Traversal Methods

A tree can be traversed and modified with the following methods:

- childrenNames()– Returns the names of the child nodes.
- parent() – Returns the parent node.
- removeNode() – Removes this node and all its descendants.
- node(String) – Returns a Preferences object, which is created if it does not already exist. The parameter is an absolute or relative path.
- nodeExists(String) – Returns true if the Preferences object identified by the path parameter exists.

9.2.4 Properties

Each Preferences node has a set of key/value pairs called properties. These properties consist of:

- *Key* – A key is a String object and *case sensitive.*
- The name-space of these keys is separate from that of the child nodes. A Preferences node could have both a child node named fudd and a property named fudd.
- *Value* – A value can always be stored and retrieved as a String object. Therefore, it must be possible to encode/decode all values into/from String objects (though it is not required to store them as such, an implementation is free to store and retrieve the value in any possible way as long as the String semantics are maintained). A number of methods are available to store and retrieve values as primitive types. These methods are provided both for the convenience of the user of the Preferences interface, and to allow an implementation the option of storing the values in a more compact form.

All the keys that are defined in a Preferences object can be obtained with the keys() method. The clear() method can be used to clear all properties from a Preferences object. A single property can be removed with the remove(String) method.

9.2.5 Storing and Retrieving Properties

The Preferences interface has a number of methods for storing and retrieving property values based on their key. All the put* methods take as parameters a key and a value. All the get* methods take as parameters a key and a default value.

- put(String,String), get(String,String)
- putBoolean(String,boolean), getBoolean(String,boolean)
- putInt(String,int), getInt(String,int)
- putLong(String,long), getLong(String,long)
- putFloat(String,float), getFloat(String,float)
- putDouble(String,double), getDouble(String,double)
- putByteArray(String,byte[]), getByteArray(String,byte[])

The methods act as if all the values are stored as String objects, even though implementations may use different representations for the different types. For example, a property can be written as a String object and read back as a float, providing that the string can be parsed as a valid Java float object. In the event of a parsing error, the get* methods do not raise exceptions, but instead return their default parameters.

9.2.6 Defaults

All get* methods take a default value as a parameter. The reasons for having such a default are:

- When a property for a Preferences object has not been set, the default is returned instead. In most cases, the bundle developer does not have to distinguish whether or not a property exists.
- A *best effort* strategy has been a specific design choice for this specification. The bundle developer should not have to react when the back-end store is not available. In those cases, the default value is returned without further notice.
  Bundle developers who want to assure that the back-end store is available should call the flush or sync method. Either of these methods will throw a BackingStoreException if the back-end store is not available.

9.3 Concurrency

This specification specifically allows an implementation to modify Preferences objects in a back-end store. If the back-end store is shared by multiple processes, concurrent updates may cause differences between the back-end store and the in-memory Preferences objects.

Bundle developers can partly control this concurrency with the flush() and sync() method. Both methods operate on a Preferences object.

The flush method performs the following actions:

- Stores (makes persistent) any ancestors (including the current node) that do not exist in the persistent store.
- Stores any properties which have been modified in this node since the last time it was flushed.
- Removes from the persistent store any child nodes that were removed from this object since the last time it was flushed.
- Flushes all existing child nodes.

The sync method will first flush, and then ensure that any changes that have been made to the current node and its descendents in the back-end store (by some other process) take effect. For example, it could fetch all the descendants into a local cache, or it could clear all the descendants from the cache so that they will be read from the back-end store as required.

If either method fails, a BackingStoreException is thrown.

The flush or sync methods provide no atomicity guarantee. When updates to the same back-end store are done concurrently by two different processes, the result may be that changes made by different processes are intermingled. To avoid this problem, implementations may simply provide a dedicated section (or name-space) in the back-end store for each OSGi environment, so that clashes do not arise, in which case there is no reason for bundle programmers to ever call sync.

In cases where sync is used, the bundle programmer needs to take into account that changes from different processes may become intermingled, and the level of granularity that can be assumed is the individual property level. Hence, for example, if two properties need to be kept in lockstep, so that one should not be changed without a corresponding change to the other, consider combining them into a single property, which would then need to be parsed into its two constituent parts.

9.4 PreferencesService Interface

The PreferencesService is obtained from the Framework's service registry in the normal way. Its purpose is to provide access to Preferences root nodes.

A Preferences Service maintains a system root and a number of user roots. User roots are automatically created, if necessary, when they are requested. Roots are maintained on a per bundle basis. For example, a user root called elmer in one bundle is distinct from a user root with the same name in another bundle. Also, each bundle has its own system root. Implementations should use a ServiceFactory service object to create a separate PreferencesService object for each bundle.

The precise description of *user* and *system* will vary from one bundle to another. The Preference Service only provides a mechanism, the bundle may use this mechanism in any desired way.

The PreferencesService interface has the following methods to access the system root and user roots:

- getSystemPreferences() – Return a Preferences object that is the root of the system preferences tree.
- getUserPreferences(String) – Return a Preferences object associated with the user name that is given as argument. If the user does not exist, a new root is created atomically.
- getUsers() – Return an array of the names of all the users for whom a Preferences tree exists.

9.5 Cleanup

The Preferences Service must listen for bundle uninstall events, and remove all the preferences data for the bundle that is being uninstalled.

It also must handle the possibility of a bundle getting uninstalled while the Preferences Service is stopped. Therefore, it must check on startup whether preferences data exists for any bundle which is not currently installed. If it does, that data must be removed.

9.6 Changes

- Added several exception clauses to the methods.
- Removed the description of JSR 10 from this specification.

9.7 org.osgi.service.prefs

The OSGi Preferences Service Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.prefs; specification-ver-
sion=1.0
```

9.7.1 Summary

- BackingStoreException - Thrown to indicate that a preferences operation could not complete because of a failure in the backing store, or a failure to contact the backing store. [p.134]
- Preferences - A node in a hierarchical collection of preference data. [p.134]
- PreferencesService - The Preferences Service. [p.144]

9.7.2 public class BackingStoreException extends Exception

Thrown to indicate that a preferences operation could not complete because of a failure in the backing store, or a failure to contact the backing store.

9.7.2.1 public BackingStoreException(String s)

s the detail message.

☐ Constructs a `BackingStoreException` with the specified detail message.

9.7.3 public interface Preferences

A node in a hierarchical collection of preference data.

This interface allows applications to store and retrieve user and system preference data. This data is stored persistently in an implementation-dependent backing store. Typical implementations include flat files, OS-specific registries, directory servers and SQL databases.

For each bundle, there is a separate tree of nodes for each user, and one for system preferences. The precise description of “user” and “system” will vary from one bundle to another. Typical information stored in the user preference tree might include font choice, and color choice for a bundle which interacts with the user via a servlet. Typical information stored in the system preference tree might include installation data, or things like high score information for a game program.

Nodes in a preference tree are named in a similar fashion to directories in a hierarchical file system. Every node in a preference tree has a *node name* (which is not necessarily unique), a unique *absolute path name*, and a path name *relative* to each ancestor including itself.

The root node has a node name of the empty `String` object (""). Every other node has an arbitrary node name, specified at the time it is created. The only restrictions on this name are that it cannot be the empty string, and it cannot contain the slash character ('/').

The root node has an absolute path name of "`/`". Children of the root node have absolute path names of "`/`" `+` *<node name>*. All other nodes have absolute path names of *<parent's absolute path name>* `+` "`/`" `+` *<node name>*. Note that all absolute path names begin with the slash character.

A node *n*'s path name relative to its ancestor *a* is simply the string that must be appended to *a*'s absolute path name in order to form *n*'s absolute path name, with the initial slash character (if present) removed. Note that:

- No relative path names begin with the slash character.
- Every node's path name relative to itself is the empty string.
- Every node's path name relative to its parent is its node name (except for the root node, which does not have a parent).
- Every node's path name relative to the root is its absolute path name with the initial slash character removed.

Note finally that:

- No path name contains multiple consecutive slash characters.
- No path name with the exception of the root's absolute path name end in the slash character.
- Any string that conforms to these two rules is a valid path name.

Each `Preference` node has zero or more properties associated with it, where a property consists of a name and a value. The bundle writer is free to choose any appropriate names for properties. Their values can be of type `String`, `long`,`int`,`boolean`, `byte[]`,`float`, or `double` but they can always be accessed as if they were `String` objects.

All node name and property name comparisons are case-sensitive.

All of the methods that modify preference data are permitted to operate asynchronously; they may return immediately, and changes will eventually propagate to the persistent backing store, with an implementation-dependent delay. The `flush` method may be used to synchronously force updates to the backing store.

Implementations must automatically attempt to flush to the backing store any pending updates for a bundle's preferences when the bundle is stopped or otherwise ungets the Preferences Service.

The methods in this class may be invoked concurrently by multiple threads in a single Java Virtual Machine (JVM) without the need for external synchronization, and the results will be equivalent to some serial execution. If this class is used concurrently *by multiple JVMs* that store their preference data in the same backing store, the data store will not be corrupted, but no other guarantees are made concerning the consistency of the preference data.

9.7.3.1 public String absolutePath( )

□ Returns this node's absolute path name. Note that:

- Root node - The path name of the root node is "/".
- Slash at end - Path names other than that of the root node may not end in slash ('/').
- Unusual names -"." and ".." have *no* special significance in path names.
- Illegal names - The only illegal path names are those that contain multiple consecutive slashes, or that end in slash and are not the root.

Returns this node's absolute path name.

9.7.3.2 public String[] childrenNames( ) throws BackingStoreException

□ Returns the names of the children of this node. (The returned array will be of size zero if this node has no children and not `null`!)

Returns the names of the children of this node.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

9.7.3.3 public void clear( ) throws BackingStoreException

□ Removes all of the properties (key-value associations) in this node. This call has no effect on any descendants of this node.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

See Also `remove(String)`[p.144]

9.7.3.4 public void flush( ) throws BackingStoreException

□ Forces any changes in the contents of this node and its descendants to the persistent store.

Once this method returns successfully, it is safe to assume that all changes made in the subtree rooted at this node prior to the method invocation have become permanent.

Implementations are free to flush changes into the persistent store at any time. They do not need to wait for this method to be called.

When a flush occurs on a newly created node, it is made persistent, as are any ancestors (and descendants) that have yet to be made persistent. Note however that any properties value changes in ancestors are *not* guaranteed to be made persistent.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

See Also sync() [p.144]

9.7.3.5 public String get(String key, String def)

key key whose associated value is to be returned.

def the value to be returned in the event that this node has no value associated with key or the backing store is inaccessible.

□ Returns the value associated with the specified key in this node. Returns the specified default if there is no value associated with the key, or the backing store is inaccessible.

Returns the value associated with key, or def if no value is associated with key.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

NullPointerException – if key is null. (A null default *is* permitted.)

9.7.3.6 public boolean getBoolean(String key, boolean def)

key key whose associated value is to be returned as a boolean.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a boolean or the backing store is inaccessible.

□ Returns the boolean value represented by the String object associated with the specified key in this node. Valid strings are "true", which represents true, and "false", which represents false. Case is ignored, so, for example, "TRUE" and "False" are also valid. This method is intended for use in conjunction with the putBoolean[p.141] method.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is something other than "true" or "false", ignoring case.

Returns the boolean value represented by the String object associated with key in this node, or null if the associated value does not exist or cannot be interpreted as a boolean.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also get(String, String) [p.137] , putBoolean(String, boolean) [p.141]

9.7.3.7 public byte[] getByteArray(String key, byte[] def)

key key whose associated value is to be returned as a byte[] object.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a byte[] type, or the backing store is inaccessible.

□ Returns the byte[] value represented by the String object associated with the specified key in this node. Valid String objects are *Base64* encoded binary data, as defined in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt), Section 6.8, with one minor change: the string must consist solely of characters from the *Base64 Alphabet*; no newline characters or extraneous characters are permitted. This method is intended for use in conjunction with the putByteArray[p.142] method.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is not a valid Base64 encoded byte array (as defined above).

Returns the byte[] value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a byte[].

Throws NullPointerException – if key is null. (A null value for def*is* permitted.)

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also get(String, String)[p.137], putByteArray(String, byte[])[p.142]

9.7.3.8 public double getDouble(String key, double def)

key key whose associated value is to be returned as a double value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a double type or the backing store is inaccessible.

□ Returns the double value represented by the String object associated with the specified key in this node. The String object is converted to a double value as by Double.parseDouble(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Double.parseDouble(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putDouble[p.142] method.

Returns the double value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a double type.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the the removeNode()[p.144] method.

NullPointerException – if key is null.

See Also putDouble(String, double)[p.142], get(String, String)[p.137]

9.7.3.9 public float getFloat(String key, float def)

key key whose associated value is to be returned as a float value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a float type or the backing store is inaccessible.

□ Returns the float value represented by the String object associated with the specified key in this node. The String object is converted to a float value as by Float.parseFloat(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Float.parseFloat(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putFloat[p.142] method.

Returns the float value represented by the string associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a float type.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

NullPointerException – if key is null.

See Also putFloat(String, float)[p.142], get(String, String)[p.137]

9.7.3.10 public int getInt(String key, int def)

key key whose associated value is to be returned as an int.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as an int or the backing store is inaccessible.

□ Returns the int value represented by the String object associated with the specified key in this node. The String object is converted to an int as by Integer.parseInt(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Integer.parseInt(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putInt[p.143] method.

Returns the int value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as an int type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also putInt(String, int)[p.143], get(String, String)[p.137]

9.7.3.11 public long getLong(String key, long def)

key key whose associated value is to be returned as a long value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a long type or the backing store is inaccessible.

□ Returns the long value represented by the String object associated with the specified key in this node. The String object is converted to a long as by Long.parseLong(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Long.parseLong(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putLong[p.143] method.

Returns the long value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a long type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also putLong(String, long)[p.143], get(String, String)[p.137]

9.7.3.12 public String[] keys( ) throws BackingStoreException

□ Returns all of the keys that have an associated value in this node. (The returned array will be of size zero if this node has no preferences and not null!)

Returns an array of the keys that have an associated value in this node.

Throws BackingStoreException – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

9.7.3.13 public String name( )

□ Returns this node's name, relative to its parent.

Returns this node's name, relative to its parent.

9.7.3.14 public Preferences node(String pathName)

pathName the path name of the Preferences object to return.

□ Returns a named Preferences object (node), creating it and any of its ancestors if they do not already exist. Accepts a relative or absolute pathname. Absolute pathnames (which begin with '/') are interpreted relative to the root of this node. Relative pathnames (which begin with any character other than '/') are interpreted relative to this node itself. The empty string ("") is a valid relative pathname, referring to this node itself.

If the returned node did not exist prior to this call, this node and any ancestors that were created by this call are not guaranteed to become persistent until the flush method is called on the returned node (or one of its descendants).

Returns the specified Preferences object.

Throws IllegalArgumentException – if the path name is invalid.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

NullPointerException – if path name is null.

See Also flush()[p.136]

9.7.3.15 public boolean nodeExists(String pathName) throws BackingStoreException

pathName the path name of the node whose existence is to be checked.

□ Returns true if the named node exists. Accepts a relative or absolute pathname. Absolute pathnames (which begin with '/') are interpreted relative to the root of this node. Relative pathnames (which begin with any character other than '/') are interpreted relative to this node itself. The pathname "" is valid, and refers to this node itself.

If this node (or an ancestor) has already been removed with the `removeNode()`[p.144] method, it *is* legal to invoke this method, but only with the pathname ""; the invocation will return `false`. Thus, the idiom `p.nodeExists("")` may be used to test whether `p` has been removed.

Returns true if the specified node exists.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method and `pathname` is not the empty string ("").

`IllegalArgumentException` – if the path name is invalid (i.e., it contains multiple consecutive slash characters, or ends with a slash character and is more than one character long).

9.7.3.16 public Preferences parent( )

□ Returns the parent of this node, or `null` if this is the root.

Returns the parent of this node.

Throws `IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

9.7.3.17 public void put(String key, String value)

key key with which the specified value is to be associated.

value value to be associated with the specified key.

□ Associates the specified value with the specified key in this node.

Throws `NullPointerException` – if `key` or `value` is `null`.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

9.7.3.18 public void putBoolean(String key, boolean value)

key `key` with which the string form of value is to be associated.

value value whose string form is to be associated with `key`.

□ Associates a `String` object representing the specified `boolean` value with the specified key in this node. The associated string is "true" if the value is `true`, and "false" if it is `false`. This method is intended for use in conjunction with the `getBoolean`[p.137] method.

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports `boolean` values, it is not unreasonable to use them. This implementation detail is not visible through the `Preferences` API, which allows the value to be read as a `boolean` (with `getBoolean`) or a `String` (with `get`) type.

Throws `NullPointerException` – if `key` is `null`.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getBoolean(String, boolean)[p.137], get(String, String)[p.137]

9.7.3.19 public void putByteArray(String key, byte[] value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified byte[] with the specified key in this node. The associated String object the *Base64* encoding of the byte[], as defined in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt), Section 6.8, with one minor change: the string will consist solely of characters from the *Base64 Alphabet*; it will not contain any newline characters. This method is intended for use in conjunction with the getByteArray[p.137] method.

Implementor's note: it is *not* necessary that the value be represented by a String type in the backing store. If the backing store supports byte[] values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as an a byte[] object (with getByteArray) or a String object (with get).

Throws NullPointerException – if key or value is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getByteArray(String, byte[])[p.137], get(String, String)[p.137]

9.7.3.20 public void putDouble(String key, double value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified double value with the specified key in this node. The associated String object is the one that would be returned if the double value were passed to Double.toString(double). This method is intended for use in conjunction with the getDouble[p.138] method

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports double values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a double (with getDouble) or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getDouble(String, double)[p.138]

9.7.3.21 public void putFloat(String key, float value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified float value with the specified key in this node. The associated String object is the one that would be returned if the float value were passed to Float.toString(float). This method is intended for use in conjunction with the getFloat[p.138] method.

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports float values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a float (with getFloat) or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getFloat(String, float)[p.138]

9.7.3.22 public void putInt(String key, int value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified int value with the specified key in this node. The associated string is the one that would be returned if the int value were passed to Integer.toString(int). This method is intended for use in conjunction with getInt[p.139] method.

Implementor's note: it is *not* necessary that the property value be represented by a String object in the backing store. If the backing store supports integer values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as an int (with getInt or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getInt(String, int)[p.139]

9.7.3.23 public void putLong(String key, long value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified long value with the specified key in this node. The associated String object is the one that would be returned if the long value were passed to Long.toString(long). This method is intended for use in conjunction with the getLong[p.139] method.

Implementor's note: it is *not* necessary that the value be represented by a String type in the backing store. If the backing store supports long values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a long (with getLong or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getLong(String, long) [p.139]

9.7.3.24 public void remove(String key)

key key whose mapping is to be removed from this node.

□ Removes the value associated with the specified key in this node, if any.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also get(String, String) [p.137]

9.7.3.25 public void removeNode( ) throws BackingStoreException

□ Removes this node and all of its descendants, invalidating any properties contained in the removed nodes. Once a node has been removed, attempting any method other than name(),absolutePath() or nodeExists("") on the corresponding Preferences instance will fail with an IllegalStateException. (The methods defined on Object can still be invoked on a node after it has been removed; they will not throw IllegalStateException.)

The removal is not guaranteed to be persistent until the flush method is called on the parent of this node. (It is illegal to remove the root node.)

Throws IllegalStateException – if this node (or an ancestor) has already been removed with the removeNode()[p.144] method.

RuntimeException – if this is a root node.

BackingStoreException – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

See Also flush() [p.136]

9.7.3.26 public void sync( ) throws BackingStoreException

□ Ensures that future reads from this node and its descendants reflect any changes that were committed to the persistent store (from any VM) prior to the sync invocation. As a side-effect, forces any changes in the contents of this node and its descendants to the persistent store, as if the flush method had been invoked on this node.

Throws BackingStoreException – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also flush() [p.136]

9.7.4 public interface PreferencesService

The Preferences Service.

Each bundle using this service has its own set of preference trees: one for system preferences, and one for each user.

A `PreferencesService` object is specific to the bundle which obtained it from the service registry. If a bundle wishes to allow another bundle to access its preferences, it should pass its `PreferencesService` object to that bundle.

9.7.4.1 public Preferences getSystemPreferences( )

- Returns the root system node for the calling bundle.

9.7.4.2 public Preferences getUserPreferences(String name)

- Returns the root node for the specified user and the calling bundle.

9.7.4.3 public String[] getUsers( )

- Returns the names of users for which node trees exist.

9.8 References

[20] *JSR 10 Preferences API*
http://www.jcp.org/jsr/detail/10.jsp

[21] *Windows Registry*
http://www.microsoft.com/technet/win98/reg.asp

[22] *RFC 2045 Base 64 encoding*
http://www.ietf.org/rfc/rfc2045.txt

14 XML Parser Service Specification

Version 1.0

14.1 Introduction

The Extensible Markup Language (XML) has become a popular method of describing data. As more bundles use XML to describe their data, a common XML Parser becomes necessary in an embedded environment in order to reduce the need for space. Not all XML Parsers are equivalent in function, however, and not all bundles have the same requirements on an XML parser.

This problem was addressed in the Java API for XML Processing, see [18] *JAXP* for Java 2 Standard Edition and Enterprise Edition. This specification addresses how the classes defined in JAXP can be used in an OSGi Service Platform. It defines how:

- Implementations of XML parsers can become available to other bundles
- Bundles can find a suitable parser
- A standard parser in a JAR can be transformed to a bundle

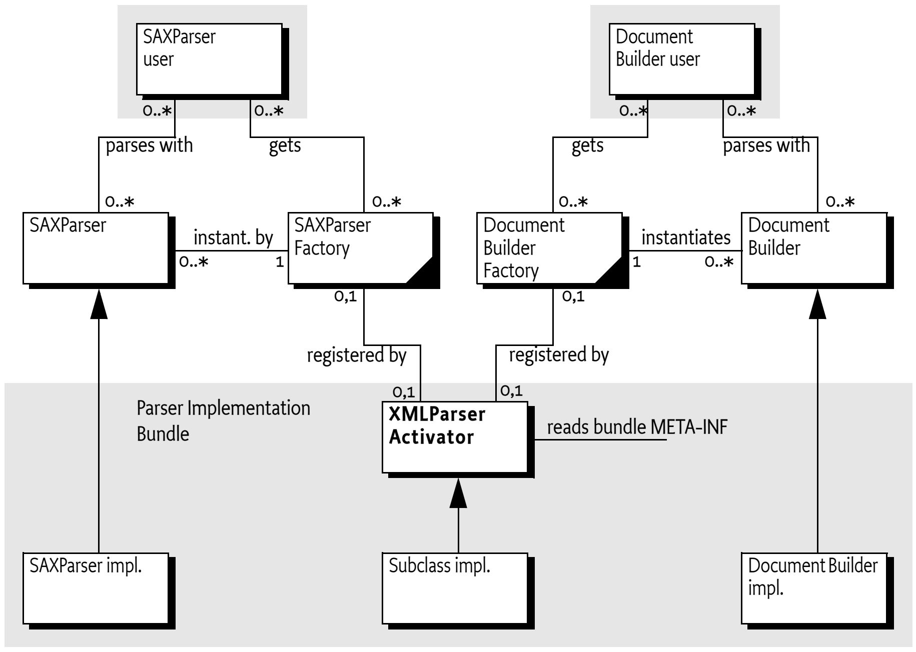
14.1.1 Essentials

- *Standards* – Leverage existing standards in Java based XML parsing: JAXP, SAX and DOM
- *Unmodified JAXP code* – Run unmodified JAXP code
- *Simple* – It should be easy to provide a SAX or DOM parser as well as easy to find a matching parser
- *Multiple* – It should be possible to have multiple implementations of parsers available
- *Extendable* – It is likely that parsers will be extended in the future with more functionality

14.1.2 Entities

- *XMLParserActivator* – A utility class that registers a parser factory from declarative information in the Manifest file.
- *SAXParserFactory* – A class that can create an instance of a `SAXParser` class.
- *DocumentBuilderFactory* – A class that can create an instance of a `DocumentBuilder` class.
- *SAXParser* – A parser, instantiated by a `SaxParserFactory` object, that parses according to the SAX specifications.
- *DocumentBuilder* – A parser, instantiated by a `DocumentBuilderFactory`, that parses according to the DOM specifications.

Figure 26 XML Parsing diagram



14.1.3 Operations

A bundle containing a SAX or DOM parser is started. This bundle registers a SAXParserFactory and/or a DocumentBuilderFactory service object with the Framework. Service registration properties describe the features of the parsers to other bundles. A bundle that needs an XML parser will get a SAXParserFactory or DocumentBuilderFactory service object from the Framework service registry. This object is then used to instantiate the requested parsers according to their specifications.

14.2 JAXP

XML has become very popular in the last few years because it allows the interchange of complex information between different parties. Though only a single XML standard exists, there are multiple APIs to XML parsers, primarily of two types:

- The Simple API for XML (SAX1 and SAX2)
- Based on the Document Object Model (DOM 1 and 2)

Both standards, however, define an abstract API that can be implemented by different vendors.

A given XML Parser implementation may support either or both of these parser types by implementing the org.w3c.dom and/or org.xml.sax packages. In addition, parsers have characteristics such as whether they are validating or non-validating parsers and whether or not they are name-space aware.

An application which uses a specific XML Parser must code to that specific parser and become coupled to that specific implementation. If the parser has implemented [18] *JAXP*, however, the application developer can code against SAX or DOM and let the runtime environment decide which parser implementation is used.

JAXP uses the concept of a *factory*. A factory object is an object that abstracts the creation of another object. JAXP defines a DocumentBuilderFactory and a SAXParserFactory class for this purpose.

JAXP is implemented in the javax.xml.parsers package and provides an abstraction layer between an application and a specific XML Parser implementation. Using JAXP, applications can choose to use any JAXP compliant parser without changing any code, simply by changing a System property which specifies the SAX- and DOM factory class names.

In JAXP, the default factory is obtained with a static method in the SAXParserFactory or DocumentBuilderFactory class. This method will inspect the associated System property and create a new instance of that class.

14.3 XML Parser service

The current specification of JAXP has the limitation that only one of each type of parser factories can be registered. This specification specifies how multiple SAXParserFactory objects and DocumentBuilderFactory objects can be made available to bundles simultaneously.

Providers of parsers should register a JAXP factory object with the OSGi service registry under the factory class name. Service properties are used to describe whether the parser:

- Is validating
- Is name-space aware
- Has additional features

With this functionality, bundles can query the OSGi service registry for parsers supporting the specific functionality that they require.

14.4 Properties

Parsers must be registered with a number of properties that qualify the service. In this specification, the following properties are specified:

- PARSER_NAMESPACEAWARE – The registered parser is aware of name-spaces. Name-spaces allow an XML document to consist of independently developed DTDs. In an XML document, they are recognized by the xmlns attribute and names prefixed with an abbreviated name-space identifier, like: <xsl:if ...>. The type is a Boolean object that must be true when the parser supports name-spaces. All other values, or the absence of the property, indicate that the parser does not implement name-spaces.
- PARSER_VALIDATING – The registered parser can read the DTD and can validate the XML accordingly. The type is a Boolean object that must

true when the parser is validating. All other values, or the absence of the property, indicate that the parser does not validate.

14.5 Getting a Parser Factory

Getting a parser factory requires a bundle to get the appropriate factory from the service registry. In a simple case in which a non-validating, non-name-space aware parser would suffice, it is best to use getServiceReference(String).

```
DocumentBuilder getParser(BundleContext context)
   throws Exception {
   ServiceReference ref = context.getServiceReference(
      DocumentBuilderFactory.class.getName() );
   if ( ref == null )
      return null;
   DocumentBuilderFactory factory =
      (DocumentBuilderFactory) context.getService(ref);
   return factory.newDocumentBuilder();
}
```

In a more demanding case, the filtered version allows the bundle to select a parser that is validating and name-space aware:

```
SAXParser getParser(BundleContext context)
   throws Exception {
   ServiceReference refs[] = context.getServiceReferences(
      SAXParserFactory.class.getName(),
         "(&(parser.namespaceAware=true)"
      +  "(parser.validating=true))" );
   if ( refs == null )
      return null;
   SAXParserFactory factory =
      (SAXParserFactory) context.getService(refs[0]);
   return factory.newSAXParser();
}
```

14.6 Adapting a JAXP Parser to OSGi

If an XML Parser supports JAXP, then it can be converted to an OSGi aware bundle by adding a BundleActivator class which registers an XML Parser Service. The utility org.osgi.util.xml.XMLParserActivator class provides this function and can be added (copied, not referenced) to any XML Parser bundle, or it can be extended and customized if desired.

14.6.1 JAR Based Services

Its functionality is based on the definition of the [19] *JAR File specification, services directory*. This specification defines a concept for service providers. A JAR file can contain an implementation of an abstractly defined service. The class (or classes) implementing the service are designated from a file in the META-INF/services directory. The name of this file is the same as the abstract service class.

The content of the UTF-8 encoded file is a list of class names separated by new lines. White space is ignored and the number sign ('#' or \u0023) is the comment character.

JAXP uses this service provider mechanism. It is therefore likely that vendors will place these service files in the META-INF/services directory.

14.6.2 XMLParserActivator

To support this mechanism, the XML Parser service provides a utility class that should be normally delivered with the OSGi Service Platform implementation. This class is a Bundle Activator and must start when the bundle is started. This class is copied into the parser bundle, and *not* imported.

The start method of the utlity BundleActivator class will look in the META-INF/services service provider directory for the files javax.xml.parsers.SAXParserFactory (SAXFACTORYNAME) or javax.xml.parsers.DocumentBuilderFactory (DOMFACTORYNAME). The full path name is specified in the constants SAXCLASSFILE and DOMCLASSFILE respectively.

If either of these files exist, the utility BundleActivator class will parse the contents according to the specification. A service provider file can contain multiple class names. Each name is read and a new instance is created. The following example shows the possible content of such a file:

```
# ACME example SAXParserFactory file
com.acme.saxparser.SAXParserFast        # Fast
com.acme.saxparser.SAXParserValidating  # Validates
```

Both the javax.xml.parsers.SAXParserFactory and the javax.xml.parsers.DocumentBuilderFactory provide methods that describe the features of the parsers they can create. The XMLParserActivator activator will use these methods to set the values of the properties, as defined in *Properties* on page 153, that describe the instances.

14.6.3 Adapting an Existing JAXP Compatible Parser

To incorporate this bundle activator into a XML Parser Bundle, do the following:

- If SAX parsing is supported, create a /META-INF/services/javax.xml.parsers.SAXParserFactory resource file containing the class names of the SAXParserFactory classes.
- If DOM parsing is supported, create a /META-INF/services/javax.xml.parsers.DocumentBuilderFactory file containing the fully qualified class names of the DocumentBuilderFactory classes.

- Create manifest file which imports the packages org.w3c.dom, org.xml.sax, and javax.xml.parsers.
- Add a Bundle-Activator header to the manifest pointing to the XMLParserActivator, the sub-class that was created, or a fully custom one.
- If the parsers support attributes, properties, or features that should be registered as properties so they can be searched, extend the XMLParserActivator class and override setSAXProperties(javax.xml.parsers.SAXParserFactory,Hashtable) and setDOMProperties(javax.xml.parsers.DocumentBuilderFactory,Hashtable).
- Ensure that custom properties are put into the Hashtable object. JAXP does not provide a way for XMLParserActivator to query the parser to find out what properties were added.
- Bundles that extend the XMLParserActivator class must call the original methods via super to correctly initialize the XML Parser Service properties.
- Compile this class into the bundle.
- Install the new XML Parser Service bundle.
- Ensure that the org.osgi.util.xml.XMLParserActivator class is is contained in the bundle.

14.7 Usage of JAXP

A single bundle should export the JAXP, SAX, and DOM APIs. The version of contained packages must be appropriately labeled. JAXP 1.1 or later is required which references SAX 2 and DOM 2. See [18] *JAXP* for the exact version dependencies.

This specification is related to related packages as defined in the JAXP 1.1 document. Table 9 contains the expected minimum versions.

Package	Minimum Version
javax.xml.parsers	1.1
org.xml.sax	2.0
org.xml.sax.helpers	2.0
org.xsml.sax.ext	1.0
org.w3c.dom	2.0

Table 9 JAXP 1.1 minimum package versions

The Xerces project from the Apache group, [20] *Xerces 2 Java Parser*, contains a number libraries that implement the necessary APIs. These libraries can be wrapped in a bundle to provide the relevant packages.

14.8 Security

A centralized XML parser is likely to see sensitive information from other bundles. Provisioning an XML parser should therefore be limited to trusted bundles. This security can be achieved by providing ServicePermission[REGISTER,javax.xml.parsers.DocumentBuilderFactory | javax.xml.parsers.SAXFactory] to only trusted bundles.

Using an XML parser is a common function, and ServicePermission[GET, javax.xml.parsers.DOMParserFactory | javax.xml.parsers.SAXFactory] should not be restricted.

The XML parser bundle will need FilePermission[<<ALL FILES>>,READ] for parsing of files because it is not known beforehand where those files will be located. This requirement further implies that the XML parser is a system bundle that must be fully trusted.

14.9 org.osgi.util.xml

The OSGi XML Parser service Package. Specification Version 1.0.

14.9.1 public class XMLParserActivator implements BundleActivator , ServiceFactory

A BundleActivator class that allows any JAXP compliant XML Parser to register itself as an OSGi parser service. Multiple JAXP compliant parsers can concurrently register by using this BundleActivator class. Bundles who wish to use an XML parser can then use the framework's service registry to locate available XML Parsers with the desired characteristics such as validating and namespace-aware.

The services that this bundle activator enables a bundle to provide are:

- javax.xml.parsers.SAXParserFactory(SAXFACTORYNAME[p.158])
- javax.xml.parsers.DocumentBuilderFactory(DOMFACTORYNAME[p.158])

The algorithm to find the implementations of the abstract parsers is derived from the JAR file specifications, specifically the Services API.

An XMLParserActivator assumes that it can find the class file names of the factory classes in the following files:

- /META-INF/services/javax.xml.parsers.SAXParserFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the SAXParserFactory.
- /META-INF/services/javax.xml.parsers.DocumentBuilderFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the DocumentBuilderFactory

If either of the files does not exist, XMLParserActivator assumes that the parser does not support that parser type.

XMLParserActivator attempts to instantiate both the SAXParserFactory and the DocumentBuilderFactory. It registers each factory with the framework along with service properties:

- PARSER_VALIDATING[p.158] - indicates if this factory supports validating parsers. It's value is a Boolean.
- PARSER_NAMESPACEAWARE[p.158] - indicates if this factory supports namespace aware parsers It's value is a Boolean.

Individual parser implementations may have additional features, properties, or attributes which could be used to select a parser with a filter. These can be added by extending this class and overriding the setSAXProperties and setDOMProperties methods.

14.9.1.1 public static final String DOMCLASSFILE = "/META-INF/services/javax.xml.parsers.DocumentBuilderFactory"

Fully qualified path name of DOM Parser Factory Class Name file

14.9.1.2 public static final String DOMFACTORYNAME = "javax.xml.parsers.DocumentBuilderFactory"

Filename containing the DOM Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

14.9.1.3 public static final String PARSER_NAMESPACEAWARE = "parser.namespaceAware"

Service property specifying if factory is configured to support namespace aware parsers. The value is of type Boolean.

14.9.1.4 public static final String PARSER_VALIDATING = "parser.validating"

Service property specifying if factory is configured to support validating parsers. The value is of type Boolean.

14.9.1.5 public static final String SAXCLASSFILE = "/META-INF/services/javax.xml.parsers.SAXParserFactory"

Fully qualified path name of SAX Parser Factory Class Name file

14.9.1.6 public static final String SAXFACTORYNAME = "javax.xml.parsers.SAXParserFactory"

Filename containing the SAX Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

14.9.1.7 public XMLParserActivator( )

14.9.1.8 public Object getService(Bundle bundle, ServiceRegistration registration)

bundle The bundle using the service.

registration The ServiceRegistration object for the service.

□ Creates a new XML Parser Factory object.

A unique XML Parser Factory object is returned for each call to this method.

The returned XML Parser Factory object will be configured for validating and namespace aware support as specified in the service properties of the specified ServiceRegistration object. This method can be overridden to configure additional features in the returned XML Parser Factory object.

Returns A new, configured XML Parser Factory object or null if a configuration error was encountered

14.9.1.9 public void setDOMProperties(DocumentBuilderFactory factory, Hashtable props)

factory - the DocumentBuilderFactory object

props - Hashtable of service properties.

□ Set the customizable DOM Parser Service Properties.

This method attempts to instantiate a validating parser and a namespaceaware parser to determine if the parser can support those features. The appropriate properties are then set in the specified props object.

This method can be overridden to add additional DOM2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

14.9.1.10 public void setSAXProperties(SAXParserFactory factory, Hashtable properties)

factory - the SAXParserFactory object

properties - the properties object for the service

□ Set the customizable SAX Parser Service Properties.

This method attempts to instantiate a validating parser and a namespaceaware parser to determine if the parser can support those features. The appropriate properties are then set in the specified properties object.

This method can be overridden to add additional SAX2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

14.9.1.11 public void start(BundleContext context) throws Exception

context The execution context of the bundle being started.

□ Called when this bundle is started so the Framework can perform the bundle-specific activities necessary to start this bundle. This method can be used to register services or to allocate any resources that this bundle needs.

This method must complete and return to its caller in a timely manner.

This method attempts to register a SAX and DOM parser with the Framework's service registry.

Throws `Exception` – If this method throws an exception, this bundle is marked as stopped and the Framework will remove this bundle's listeners, unregister all services registered by this bundle, and release all services used by this bundle.

See Also `Bundle.start`

14.9.1.12 public void stop(BundleContext context) throws Exception

context The execution context of the bundle being stopped.

□ This method has nothing to do as all active service registrations will automatically get unregistered when the bundle stops.

Throws `Exception` – If this method throws an exception, the bundle is still marked as stopped, and the Framework will remove the bundle's listeners, unregister all services registered by the bundle, and release all services used by the bundle.

See Also `Bundle.stop`

14.9.1.13 public void ungetService(Bundle bundle, ServiceRegistration registration, Object service)

bundle The bundle releasing the service.

registration The `ServiceRegistration` object for the service.

service The XML Parser Factory object returned by a previous call to the `getService` method.

□ Releases a XML Parser Factory object.

14.10 References

[15] *XML*
http://www.w3.org/XML

[16] *SAX*
http://www.saxproject.org/

[17] *DOM Java Language Binding*
http://www.w3.org/TR/REC-DOM-Level-1/java-language-binding.html

[18] *JAXP*
http://java.sun.com/xml/jaxp

[19] *JAR File specification, services directory*
http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html

[20] *Xerces 2 Java Parser*
http://xml.apache.org/xerces2-j

14 Initial Provisioning

Version 1.0

14.1 Introduction

To allow freedom regarding the choice of management protocol, the OSGi *Remote Management Reference Architecture* on page 29, specifies an architecture to remotely manage a Service Platform with a Management Agent. The Management Agent is implemented with a Management Bundle that can communicate with an unspecified management protocol.

This specification defines how the Management Agent can make its way to the Service Platform, and gives a structured view of the problems and their corresponding resolution methods.

The purpose of this specification is to enable the management of a Service Platform by an Operator, and (optionally) to hand over the management of the Service Platform later to another Operator. This approach is in accordance with the OSGi remote management reference architecture.

This bootstrapping process requires the installation of a Management Agent, with appropriate configuration data, in the Service Platform.

This specification consists of a prologue, in which the principles of the Initial Provisioning are outlined, and a number of mappings to different mechanisms.

14.1.1 Essentials

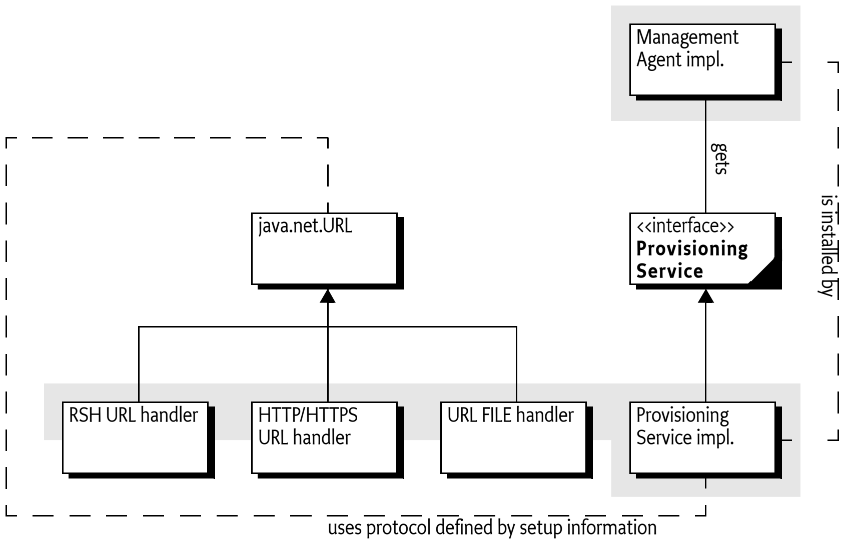
- *Policy Free* – The proposed solution must be business model agnostic; none of the affected parties (Operators, SPS Manufacturers, etc.) should be forced into any particular business model.
- *Interoperability* – The Initial Provisioning must permit arbitrary interoperability between management systems and Service Platforms. Any compliant Remote Manager should be able to manage any compliant Service Platform, even in the absence of a prior business relationship. Adhering to this requirement allows a particular Operator to manage a variety of makes and models of Service Platform Servers using a single management system of the Operator's choice. This rule also gives the consumer the greatest choice when selecting an Operator.
- *Flexible* – The management process should be as open as possible, to allow innovation and specialization while still achieving interoperability.

14.1.2 Entities

- *Provisioning Service* – A service registered with the Framework that provides information about the initial provisioning to the Management Agent.

- *Provisioning Dictionary* – A `Dictionary` object that is filled with information from the ZIP files that are loaded during initial setup.
- *RSH Protocol* – An OSGi specific secure protocol based on HTTP.
- *Management Agent* – A bundle that is responsible for managing a Service Platform under control of a Remote Manager.

Figure 37 *Initial Provisioning*



14.2 Procedure

The following procedure should be executed by an OSGi Framework implementation that supports this Initial Provisioning specification.

When the Service Platform is first brought under management control, it must be provided with an initial request URL in order to be provisioned. Either the end user or the manufacturer may provide the initial request URL. How the initial request URL is transferred to the Framework is not specified, but a mechanism might, for example, be a command line parameter when the framework is started.

When asked to start the Initial Provisioning, the Service Platform will send a request to the management system. This request is encoded in a URL, for example:

```
http://osgi.acme.com/remote-manager
```

This URL may use any protocol that is available on the Service Platform Server. Many standard protocols exist, but it is also possible to use a proprietary protocol. For example, software could be present which can communicate with a smart card and could handle, for example, this URL:

```
smart-card://com1:0/7F20/6F38
```

Before the request URL is executed, the Service Platform information is appended to the URL. This information includes at least the Service Platform Identifier, but may also contain proprietary information, as long as the keys for this information do not conflict. Different URL schemes may use different methods of appending parameters; these details are specified in the mappings of this specification to concrete protocols.

The result of the request must be a ZIP file (The content type should be application/zip). It is the responsibility of the underlying protocol to guarantee the integrity and authenticity of this ZIP file.

This ZIP file is unpacked and its entries (except bundle and bundle-url entries, described in Table 19) are placed in a Dictionary object. This Dictionary object is called the *Provisioning Dictionary*. It must be made available from the Provisioning Service in the service registry. The names of the entries in the ZIP file must not start with a slash ('/').

The ZIP file may contain only four types of dictionary entries: text, binary, bundle, or bundle-url. The types are specified in the ZIP entry's extra field, and must be a MIME type as defined in [51] *MIME Types*. The text and bundle-url entries are translated into a String object. All other entries must be stored as a byte[].

Type	MIME Type	Description
text	MIME_STRING text/plain;charset=utf-8	Must be represented as a String object
binary	MIME_BYTE_ARRAY application/octet-stream	Must be represented as a byte array (byte[]).

Table 18 *Content types of provisioning ZIP file*

Type	MIME Type	Description
bundle	MIME_BUNDLE application/x-osgi-bundle	Entries must be installed using BundleContext.installBundle(String, InputStream), with the InputStream object constructed from the contents of the ZIP entry. The location must be the name of the ZIP entry without leading slash. This entry must not be stored in the Provisioning Dictionary. If a bundle with this location name is already installed in this system, then this bundle must be updated instead of installed.
bundle-url	MIME_BUNDLE_URL text/x-osgi-bundle-url; charset=utf-8	The content of this entry is a string coded in utf-8. Entries must be installed using BundleContext.installBundle(String, InputStream), with the InputStream object created from the given URL. The location must be the name of the ZIP entry without leading slash. This entry must not be stored in the Provisioning Dictionary. If a bundle with this location url is already installed in this system, then this bundle must be updated instead of installed.

Table 18 *Content types of provisioning ZIP file*

The Provisioning Service must install (but not start) all entries in the ZIP file that are typed in the extra field with bundle or bundle-url.

If an entry named PROVISIONING_START_BUNDLE is present in the Provisioning Dictionary, then its content type must be text as defined in Table 18. The content of this entry must match the bundle location of a previously loaded bundle. This designated bundle must be given AllPermission and started.

If no PROVISIONING_START_BUNDLE entry is present in the Provisioning Dictionary, the Provisioning Dictionary should contain a reference to another ZIP file under the PROVISIONING_REFERENCE key. If both keys are absent, no further action must take place.

If this PROVISIONING_REFERENCE key is present and holds a String object that can be mapped to a valid URL, then a new ZIP file must be retrieved from this URL. The PROVISIONING_REFERENCE link may be repeated multiple times in successively loaded ZIP files.

Referring to a new ZIP file with such a URL allows a manufacturer to place a fixed reference inside the Service Platform Server (in a file or smart card) that will provide some platform identifying information and then also immediately load the information from the management system. The PROVISIONING_REFERENCE link may be repeated multiple times in successively loaded ZIP files. The entry PROVISIONING_UPDATE_COUNT must be an Integer object that must be incremented on every iteration.

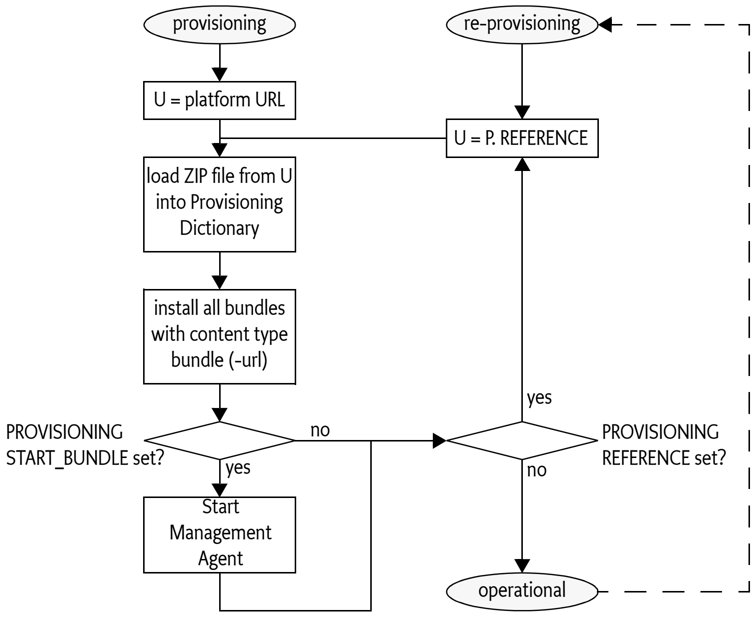
Information retrieved while loading subsequent PROVISIONING_REFERENCE URLs may replace previous key/values in the Provisioning Dictionary, but must not erase unrecognized key/values. For example, if an assignment has assigned the key proprietary-x, with a value '3', then later assignments must not override this value, unless the later loaded ZIP file contains an entry with that name. All these updates to the Provisioning Dictionary must be stored persistently. At the same time, each entry of type bundle or bundle-url (see Table 18) must be installed and not started.

Once the Management Agent has been started, the Initial Provisioning service has become operational. In this state, the Initial Provisioning service must react when the Provisioning Dictionary is updated with a new PROVISIONING_REFERENCE property. If this key is set, it should start the cycle again. For example, if the control of a Service Platform needs to be transferred to another Remote Manager, the Management Agent should set the PROVISIONING_REFERENCE to the location of this new Remote Manager's Initial Provisioning ZIP file.This process is called *re-provisioning*.

If errors occur during this process, the Initial Provisioning service should try to notify the Service User of the problem.

The previous description is depicted in Figure 38 as a flow chart.

Figure 38 *Flow chart installation Management Agent bundle*


provisioning
re-provisioning
U = platform URL
U = P. REFERENCE
load ZIP file from U into Provisioning Dictionary
install all bundles with content type bundle (-url)
PROVISIONING START_BUNDLE set?
no
yes
Start Management Agent
yes
PROVISIONING REFERENCE set?
no
operational

The Management Agent may require configuration data that is specific to the Service Platform instance. If this data is available outside the Management Agent bundle, the merging of this data with the Management Agent may take place in the Service Platform. Transferring the data separately will make it possible to simplify the implementation on the server side, as it is not necessary to create *personalized* Service Platform bundles. The PROVISIONING_AGENT_CONFIG key is reserved for this purpose, but the Management Agent may use another key or mechanisms if so desired.

The PROVISIONING_SPID key must contain the Service Platform Identifier.

14.3 Special Configurations

The next section shows some examples of specially configured types of Service Platform Servers and how they are treated with the respect to the specifications in this document.

14.3.1 Branded Service Platform Server

If a Service Platform Operator is selling Service Platform Servers branded exclusively for use with their service, the provisioning will most likely be performed prior to shipping the Service Platform Server to the User. Typically the Service Platform is configured with the Dictionary entry PROVISIONING_REFERENCE pointing at a location controlled by the Operator.

Up-to-date bundles and additional configuration data must be loaded from that location at activation time. The Service Platform is probably equipped with necessary security entities, like certificates, to enable secure downloads from the Operator's URL over open networks, if necessary.

14.3.2 Non-connected Service Platform

Circumstances might exist in which the Service Platform Server has no WAN connectivity, or prefers not to depend on it for the purposes not covered by this specification.

The non-connected case can be implemented by specifying a file:// URL for the initial ZIP file (PROVISIONING_REFERENCE). That file:// URL would name a local file containing the response that would otherwise be received from a remote server.

The value for the Management Agent PROVISIONING_REFERENCE found in that file will be used as input to the load process. The PROVISIONING_REFERENCE may point to a bundle file stored either locally or remotely. No code changes are necessary for the non-connected scenario. The file:// URLs must be specified, and the appropriate files must be created on the Service Platform.

14.4 The Provisioning Service

Provisioning information is conveyed between bundles using the Provisioning Service, as defined in the ProvisioningService interface. The Provisioning Dictionary is retrieved from the ProvisioningService object using the getInformation() method. This is a read-only Dictionary object, any changes to this Dictionary object must throw an UnsupportedOperationException.

The Provisioning Service provides a number of methods to update the Provisioning Dictionary.

- addInformation(Dictionary) – Add all key/value pairs in the given Dictionary object to the Provisioning Dictionary.
- addInformation(ZipInputStream) – It is also possible to add a ZIP file to the Provisioning Service immediately. This will unpack the ZIP file and add the entries to the Provisioning Dictionary. This method must install the bundles contained in the ZIP file as described in *Procedure* on page 226.
- setInformation(Dictionary) – Set a new Provisioning Dictionary. This will remove all existing entries.

Each of these method will increment the PROVISIONING_UPDATE_COUNT entry.

14.5 Management Agent Environment

The Management Agent should be written with great care to minimize dependencies on other packages and services, as *all* services in OSGi are optional. Some Service Platforms may have other bundles pre-installed, so it is possible that there may be exported packages and services available. Mechanisms outside the current specification, however, must be used to discover these packages and services before the Management Agent is installed.

The Provisioning Service must ensure that the Management Agent is running with AllPermission. The Management Agent should check to see if the Permission Admin service is available, and establish the initial permissions as soon as possible to insure the security of the device when later bundles are installed. As the PermissionAdmin interfaces may not be present (it is an optional service), the Management Agent should export the PermissionAdmin interfaces to ensure they can be resolved.

Once started, the Management Agent may retrieve its configuration data from the Provisioning Service by getting the byte[] object that corresponds to the PROVISIONING_AGENT_CONFIG key in the Provisioning Dictionary. The structure of the configuration data is implementation specific.

The scope of this specification is to provide a mechanism to transmit the raw configuration data to the Management Agent. The Management Agent bundle may alternatively be packaged with its configuration data in the bundle, so it may not be necessary for the Management Agent bundle to use the Provisioning Service at all.

Most likely, the Management Agent bundle will install other bundles to provision the Service Platform. Installing other bundles might even involve downloading a more full featured Management Agent to replace the initial Management Agent.

14.6 Mapping To File Scheme

The file: scheme is the simplest and most completely supported scheme which can be used by the Initial Provisioning specification. It can be used to store the configuration data and Management Agent bundle on the Service Platform Server, and avoids any outside communication.

If the initial request URL has a file scheme, no parameters should be appended, because the file: scheme does not accept parameters.

14.6.1 Example With File Scheme

The manufacturer should prepare a ZIP file containing only one entry named PROVISIONING_START_BUNDLE that contains a location string of an entry of type application/x-osgi-bundle or application/x-osgi-bundle-URL. For example, the following ZIP file demonstrates this:

```
provisioning.start.bundle  text       agent
agent                      bundle     C0AF0E9B2AB..
```

The bundle may also be specified with a URL:

```
provisioning.start.bundle  text       http://acme.com/a.jar
agent                      bundle-url http://acme.com/a.jar
```

Upon startup, the framework is provided with the URL with the file: scheme that points to this ZIP file:

```
file:/opt/osgi/ma.zip
```

14.7 Mapping To HTTP(S) Scheme

This section defines how HTTP and HTTPS URLs must be used with the Initial Provisioning specification.

- HTTP – May be used when the data exchange takes place over networks that are secured by other means, such as a Virtual Private Network (VPN) or a physically isolated network. Otherwise, HTTP is not a valid scheme because no authentication takes place.
- HTTPS – May be used if the Service Platform is equipped with appropriate certificates.

HTTP and HTTPS share the following qualities:

- Both are well known and widely used
- Numerous implementations of the protocols exist
- Caching of the Management Agent will be desired in many implementations where limited bandwidth is an issue. Both HTTP and HTTPS already contain an accepted protocol for caching.

Both HTTP and HTTPS must be used with the GET method. The response is a ZIP file, implying that the response header Content-Type header must contain application/zip.

14.7.1 HTTPS Certificates

In order to use HTTPS, certificates must be in place. These certificates, that are used to establish trust towards the Operator, may be made available to the Service Platform using the Provisioning Service. The root certificate should be assigned to the Provisioning Dictionary before the HTTPS provider is used. Additionally, the Service Platform should be equipped with a Service Platform certificate that allows the Service Platform to properly authenticate itself towards the Operator. This specification does not state how this certificate gets installed into the Service Platform.

The root certificate is stored in the Provisioning Dictionary under the key:

PROVISIONING_ROOTX509

The Root X.509 Certificate holds certificates used to represent a handle to a common base for establishing trust. The certificates are typically used when authenticating a Remote Manager to the Service Platform. In this case, a Root X.509 certificate must be part of a certificate chain for the Operator's certificate. The format of the certificate is defined in *Certificate Encoding* on page 233.

14.7.2 Certificate Encoding

Root certificates are X.509 certificates. Each individual certificate is stored as a byte[] object. This byte[] object is encoded in the default Java manner, as follows:

- The original, binary certificate data is DER encoded
- The DER encoded data is encoded into base64 to make it text.
- The base64 encoded data is prefixed with
  -----BEGIN CERTIFICATE-----
  and suffixed with:
  -----END CERTIFICATE-----
- If a record contains more than one certificate, they are simply appended one after the other, each with a delimiting prefix and suffix.

The decoding of such a certificate may be done with the java.security.cert.CertificateFactory class:

```
InputStream bis = new ByteArrayInputStream(x509); // byte[]
CertificateFactory cf =
   CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(bis);
Iterator i = c.iterator();
while (i.hasNext()) {
   Certificate cert = (Certificate)i.next();
   System.out.println(cert);
}
```

14.7.3 URL Encoding

The URL must contain the Service Platform Identity, and may contain more parameters. These parameters are encoded in the URL according to the HTTP(S) URL scheme. A base URL may be set by an end user but the Provisioning Service must add the Service Platform Identifier.

If the request URL already contains HTTP parameters (if there is a '?' in the request), the `service_platform_id` is appended to this URL as an additional parameter. If, on the other hand, the request URL does not contain any HTTP parameters, the `service_platform_id` will be appended to the URL after a '?', becoming the first HTTP parameter. The following two examples show these two variants:

```
http://server.operator.com/service-x? «
   foo=bar&service_platform_id=VIN:123456789

http://server.operator.com/service-x? «
   service_platform_id=VIN:123456789
```

Proper URL encoding must be applied when the URL contains characters that are not allowed. See [50] *RFC 2396 - Uniform Resource Identifier (URI).*

14.8 Mapping To RSH Scheme

The RSH protocol is an OSGi-specific protocol, and is included in this specification because it is optimized for Initial Provisioning. It requires a shared secret between the management system and the Service Platform that is small enough to be entered by the Service User.

RSH bases authentication and encryption on Message Authentication Codes (MACs) that have been derived from a secret that is shared between the Service Platform and the Operator prior to the start of the protocol execution.

The protocol is based on an ordinary HTTP GET request/response, in which the request must be *signed* and the response must be *encrypted* and *authenticated*. Both the *signature* and *encryption key* are derived from the shared secret using Hashed Message Access Codes (HMAC) functions.

As additional input to the HMAC calculations, one client-generated nonce and one server-generated nonce are used to prevent replay attacks. The nonces are fairly large random numbers that must be generated in relation to each invocation of the protocol, in order to guarantee freshness. These nonces are called `clientfg` (client-generated freshness guarantee) and `serverfg` (server-generated freshness guarantee).

In order to separate the HMAC calculations for authentication and encryption, each is based on a different constant value. These constants are called the *authentication constant* and the *encryption constant.*

From an abstract perspective, the protocol may be described as follows.

- δ – Shared secret, 160 bits or more
- s – Server nonce, called `servercfg`, 128 bits
- c – Client nonce, called `clientfg`, 128 bits
- K_a – Authentication key, 160 bits
- K_e – Encryption key, 192 bits
- r – Response data
- e – Encrypted data
- E – Encryption constant, a `byte[]` of 05, 36, 54, 70, 00 (hex)
- A – Authentication constant, a `byte[]` of 00, 4f, 53, 47, 49 (hex)
- M – Message material, used for K_e calculation.

- *m* – The calculated message authentication code.
- *3DES* – Triple DES, encryption function, see [52] *3DES*. The bytes of the key must be set to odd parity. CBC mode must be used where the padding method is defined in [53] *RFC 1423 Part III: Algorithms, Modes, and Identifiers*. In [55] *Java Cryptography API (part of Java 1.4)* this is addressed as `PKCS5Padding`.
- *IV* – Initialization vector for 3DES.
- *SHA1* – Secure Hash Algorithm to generate the Hashed Message Autentication Code, see [56] *SHA-1*. The function takes a single parameter, the block to be worked upon.
- *HMAC* – The fuction that calculates a message authentication code, which must HMAC-SHA1. HMAC-SHA1 is defined in [45] *HMAC: Keyed-Hashing for Message Authentication*. The HMAC function takes a key and a block to be worked upon as arguments. Note that the lower 16 bytes of the result must be used.
- *{}* – Concatenates its arguments
- *[]* – Indicates access to a sub-part of a variable, in bytes. Index starts at one, not zero.

In each step, the emphasized server or client indicates the context of the calculation. If both are used at the same time, each variable will have server or client as a subscript.

1. The *client* generates a random nonce, stores it and denotes it `clientfg`

   $c = nonce$

2. The client sends the request with the `clientfg` to the server.

   $c_{server} \Leftarrow c_{client}$

3. The *server* generates a nonce and denotes it `serverfg`.

   $s = nonce$

4. The *server* calculates an authentication key based on the SHA1 function, the shared secret, the received `clientfg`, the `serverfg` and the authentication constant.

   $K_a \leftarrow SHA1(\{\delta, c, s, A\})$

5. The *server* calculates an encryption key using an SHA-1 function, the shared secret, the received `clientfg`, the `serverfg` and the encryption constant. It must first calculate the *key material* M.

   $M[1, 20] \leftarrow SHA1(\{\delta, c, s, E\})$

   $M[21, 40] \leftarrow SHA1(\{\delta, M[1, 20], c, s, E\})$

6. The key for DES consists K_e and IV.

   $K_e \leftarrow M[1, 24]$

   $IV \leftarrow M[25, 32]$

   The *server* encrypts the response data using the encryption key derived in 5. The encryption algorithm that must be used to encrypt/decrypt the response data is 3DES. 24 bytes (192 bits) from M are used to generate K_e, but the low order bit of each byte must be used as an odd parity bit. This means that before using K_e, each byte must be processed to set the low order bit so that the byte has odd parity.

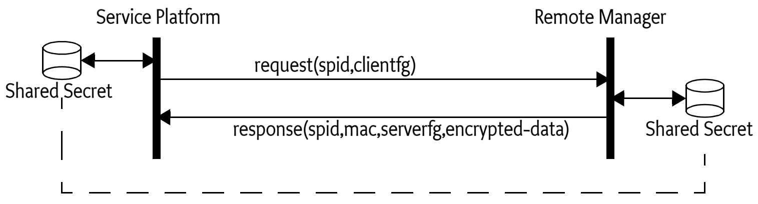
The encryption/decryption key used is specified by the following:
$e \leftarrow 3DES(K_e, IV, r)$

7. The *server* calculates a MAC *m* using the HMAC function, the encrypted response data and the authentication key derived in 4.
$m \leftarrow HMAC(K_a, e)$

8. The *server* sends a response to the *client* containing the `serverfg`, the MAC *m* and the encrypted response data
$s_{client} \Leftarrow s_{server}$
$m_{client} \Leftarrow m_{server}$
$e_{client} \Leftarrow e_{server}$
The *client* calculates the encryption key K_e the same way the server did in step 5 and 6. and uses this to decrypt the encrypted response data. The `serverfg` value received in the response is used in the calculation.
$r \leftarrow 3DES(K_e, IV, e)$

9. The *client* performs the calculation of the MAC *m'* in the same way the server did, and checks that the results match the received MAC *m*. If they do not match, further processing is discarded. The `serverfg` value received in the response is used in the calculation.
$K_a \leftarrow SHA1(\{\delta, c, s, A\})$
$m' \leftarrow HMAC(K_a, e)$
$m' = m$

Figure 39 *Action Diagram for RSH*



14.8.1 Shared Secret

The *shared secret* should be a key of length 160 bits (20 bytes) or more. The length is selected to match the output of the selected hash algorithm [46] *NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.*.

In some scenarios, the shared secret is generated by the Operator and communicated to the User, who inserts the secret into the Service Platform through some unspecified means.

The opposite is also possible: the shared secret can be stored within the Service Platform, extracted from it, and then communicated to the Operator. In this scenario, the source of the shared secret could be either the Service Platform or the Operator.

In order for the server to calculate the authentication and encryption keys, it requires the proper shared secret. The server must have access to many different shared secrets, one for each Service Platform it is to support. To be able to resolve this issue, the server must typically also have access to the Service Platform Identifier of the Service Platform. The normal way for the server to know the Service Platform Identifier is through the application protocol, as this value is part of the URL encoded parameters of the HTTP, HTTPS, or RSH mapping of the Initial Provisioning.

In order to be able to switch Operators, a new shared secret must be used. The new secret may be generated by the new Operator and then inserted into the Service Platform device using a mechanism not covered by this specification. Or the device itself may generate the new secret and convey it to the owner of the device using a display device or read-out, which is then communicated to the new operator out-of-band. Additionally, the generation of the new secret may be triggered by some external event, like holding down a button for a specified amount of time.

14.8.2 Request Coding

RSH is mapped to HTTP or HTTPS. Thus, the request parameters are URL encoded as discussed in 14.7.3 *URL Encoding*. RSH requires an additional parameter in the URL: the `clientfg` parameter. This parameter is a nonce that is used to counter replay attacks. See also *RSH Transport* on page 238.

14.8.3 Response Coding

The server's response to the client is composed of three parts:

- A header containing the protocol version and the `serverfg`
- The MAC
- The encrypted response

These three items are packaged into a binary container according to Table 19.

Bytes	Description	Value hex
4	Number of bytes in header	2E
1	Major version number	01
1	Minor version number	00
16	`serverfg`	...
4	Number of bytes in MAC	10
16	Message Authentication Code	MAC
4	Number of bytes of encrypted ZIP file	N
N	Encrypted ZIP file	...

Table 19 RSH Header description

The response content type is an RSH-specific encrypted ZIP file, implying that the response header Content-Type must be application/x-rsh for the HTTP request. When the content file is decrypted, the content must be a ZIP file.

14.8.4 RSH URL

The RSH URL must be used internally within the Service Platform to indicate the usage of RSH for initial provisioning. The RSH URL format is identical to the HTTP URL format, except that the scheme is rsh: instead of http:. For example (« means line continues on next line):

```
rsh://server.operator.com/service-x
```

14.8.5 Extensions to the Provisioning Service Dictionary

RSH specifies one additional entry for the Provisioning Dictionary:

PROVISIONING_RSH_SECRET

The value of this entry is a byte[] containing the shared secret used by the RSH protocol.

14.8.6 RSH Transport

RSH is mapped to HTTP or HTTPS and follows the same URL encoding rules, except that the clientfg is additionally appended to the URL. The key in the URL must be clientfg and the value must be encoded in base 64 format:

The clientfg parameter is transported as an HTTP parameter that is appended after the service_platform_id parameter. The second example above would then be:

```
rsh://server.operator.com/service-x
```

Which, when mapped to HTTP, must become:

```
http://server.operator.com/service-x? «
   service_platform_id=VIN:123456789& «
   clientfg=AHPmWcw%2FsiWYC37xZNdKvQ%3D%3D
```

14.9 Security

The security model for the Service Platform is based on the integrity of the Management Agent deployment. If any of the mechanisms used during the deployment of management agents are weak, or can be compromised, the whole security model becomes weak.

From a security perspective, one attractive means of information exchange would be a smart card. This approach enables all relevant information to be stored in a single place. The Operator could then provide the information to the Service Platform by inserting the smart card into the Service Platform.

14.9.1 Concerns

The major security concerns related to the deployment of the Management Agent are:

- The Service Platform is controlled by the intended Operator
- The Operator controls the intended Service Platform(s)
- The integrity and confidentiality of the information exchange that takes place during these processes must be considered

In order to address these concerns, an implementation of the OSGi Remote Management Architecture must assure that:

- The Operator authenticates itself to the Service Platform
- The Service Platform authenticates itself to the Operator
- The integrity and confidentiality of the Management Agent, certificates, and configuration data are fully protected if they are transported over public transports.

Each mapping of the Initial Provisioning specification to a concrete implementation must describe how these goals are met.

14.9.2 Service Platform Long-Term Security

Secrets for long-term use may be exchanged during the Initial Provisioning procedures. This way, one or more secrets may be shared securely, assuming that the Provisioning Dictionary assignments used are implemented with the proper security characteristics.

14.9.3 Permissions

The provisioning information may contain sensitive information. Also, the ability to modify provisioning information can have drastic consequences. Thus, only trusted bundles should be allowed to register, or get the Provisioning Service. This restriction can be enforced using `ServicePermission[GET, ProvisioningService]`.

No `Permission` classes guard reading or modification of the Provisioning Dictionary, so care must be taken not to leak the `Dictionary` object received from the Provisioning Service to bundles that are not trusted.

Whether message-based or connection-based, the communications used for Initial Provisioning must support mutual authentication and message integrity checking, at a minimum.

By using both server and client authentication in HTTPS, the problem of establishing identity is solved. In addition, HTTPS will encrypt the transmitted data. HTTPS requires a Public Key Infrastructure implementation in order to retrieve the required certificates.

When RSH is used, it is vital that the shared secret is shared only between the Operator and the Service Platform, and no one else.

14.10 org.osgi.service.provisioning

The OSGi Provisioning Service Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.provisioning; specification-
version=1.0
```

14.10.1 public interface ProvisioningService

Service for managing the initial provisioning information.

Initial provisioning of an OSGi device is a multi step process that culminates with the installation and execution of the initial management agent. At each step of the process, information is collected for the next step. Multiple bundles may be involved and this service provides a means for these bundles to exchange information. It also provides a means for the initial Management Bundle to get its initial configuration information.

The provisioning information is collected in a `Dictionary` object, called the Provisioning Dictionary. Any bundle that can access the service can get a reference to this object and read and update provisioning information. The key of the dictionary is a `String` object and the value is a `String` or `byte[]` object. The single exception is the PROVISIONING_UPDATE_COUNT value which is an Integer. The `provisioning` prefix is reserved for keys defined by OSGi, other key names may be used for implementation dependent provisioning systems.

Any changes to the provisioning information will be reflected immediately in all the dictionary objects obtained from the Provisioning Service.

Because of the specific application of the Provisioning Service, there should be only one Provisioning Service registered. This restriction will not be enforced by the Framework. Gateway operators or manufactures should ensure that a Provisioning Service bundle is not installed on a device that already has a bundle providing the Provisioning Service.

The provisioning information has the potential to contain sensitive information. Also, the ability to modify provisioning information can have drastic consequences. Thus, only trusted bundles should be allowed to register and get the Provisioning Service. The `ServicePermission` is used to limit the bundles that can gain access to the Provisioning Service. There is no check of `Permission` objects to read or modify the provisioning information, so care must be taken not to leak the Provisioning Dictionary received from `getInformation` method.

14.10.1.1 public static final String MIME_BUNDLE = “application/x-osgi-bundle”

MIME type to be stored in the extra field of a `ZipEntry` object for an installable bundle file. Zip entries of this type will be installed in the framework, but not started. The entry will also not be put into the information dictionary.

14.10.1.2 public static final String MIME_BUNDLE_URL = “text/x-osgi-bundle-url”

MIME type to be stored in the extra field of a ZipEntry for a String that represents a URL for a bundle. Zip entries of this type will be used to install (but not start) a bundle from the URL. The entry will not be put into the information dictionary.

14.10.1.3 public static final String MIME_BYTE_ARRAY = “application/octet-stream”

MIME type to be stored in the extra field of a `ZipEntry` object for `byte[]` data.

14.10.1.4 public static final String MIME_STRING = "text/plain;charset=utf-8"

MIME type to be stored in the extra field of a `ZipEntry` object for String data.

14.10.1.5 public static final String PROVISIONING_AGENT_CONFIG = "provisioning.agent.config"

The key to the provisioning information that contains the initial configuration information of the initial Management Agent. The value will be of type `byte[]`.

14.10.1.6 public static final String PROVISIONING_REFERENCE = "provisioning.reference"

The key to the provisioning information that contains the location of the provision data provider. The value must be of type `String`.

14.10.1.7 public static final String PROVISIONING_ROOTX509 = "provisioning.rootx509"

The key to the provisioning information that contains the root X509 certificate used to esatblish trust with operator when using HTTPS.

14.10.1.8 public static final String PROVISIONING_RSH_SECRET = "provisioning.rsh.secret"

The key to the provisioning information that contains the shared secret used in conjunction with the RSH protocol.

14.10.1.9 public static final String PROVISIONING_SPID = "provisioning.spid"

The key to the provisioning information that uniquely identifies the Service Platform. The value must be of type `String`.

14.10.1.10 public static final String PROVISIONING_START_BUNDLE = "provisioning.start.bundle"

The key to the provisioning information that contains the location of the bundle to start with `AllPermission`. The bundle must have be previously installed for this entry to have any effect.

14.10.1.11 public static final String PROVISIONING_UPDATE_COUNT = "provisioning.update.count"

The key to the provisioning information that contains the update count of the info data. Each set of changes to the provisioning information must end with this value being incremented. The value must be of type `Integer`. This key/value pair is also reflected in the properties of the ProvisioningService in the service registry.

14.10.1.12 public void addInformation(Dictionary info)

info the set of Provisioning Information key/value pairs to add to the Provisioning Information dictionary. Any keys are values that are of an invalid type will be silently ignored.

□ Adds the key/value pairs contained in `info` to the Provisioning Information dictionary. This method causes the `PROVISIONING_UPDATE_COUNT` to be incremented.

14.10.1.13 public void addInformation(ZipInputStream zis) throws IOException

zis the `ZipInputStream` that will be used to add key/value pairs to the Provisioning Information dictionary and install and start bundles. If a `ZipEntry` does not have an `Extra` field that corresponds to one of the four defined MIME types (`MIME_STRING`, `MIME_BYTE_ARRAY`,`MIME_BUNDLE`, and `MIME_BUNDLE_URL`) in will be silently ignored.

□ Processes the `ZipInputStream` and extracts information to add to the Provisioning Information dictionary, as well as, install/update and start bundles. This method causes the `PROVISIONING_UPDATE_COUNT` to be incremented.

Throws `IOException` – if an error occurs while processing the ZipInputStream. No additions will be made to the Provisioning Information dictionary and no bundles must be started or installed.

14.10.1.14 public Dictionary getInformation( )

□ Returns a reference to the Provisioning Dictionary. Any change operations (put and remove) to the dictionary will cause an `UnsupportedOperationException` to be thrown. Changes must be done using the `setInformation` and `addInformation` methods of this service.

14.10.1.15 public void setInformation(Dictionary info)

info the new set of Provisioning Information key/value pairs. Any keys are values that are of an invalid type will be silently ignored.

□ Replaces the Provisioning Information dictionary with the key/value pairs contained in `info`. Any key/value pairs not in `info` will be removed from the Provisioning Information dictionary. This method causes the `PROVISIONING_UPDATE_COUNT` to be incremented.

14.11 References

[45] *HMAC:* Keyed-Hashing for Message Authentication http://www.ietf.org/rfc/rfc2104.txt Krawczyk ,et. al. 1997.

[46] *NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.*

[47] *Hypertext Transfer Protocol - HTTP/1.1* http://www.ietf.org/rfc/rfc2616.txt *Fielding, R., et. al.*

[48] *Rescorla, E., HTTP over TLS, IETF RFC 2818, May 2000* http://www.ietf.org/rfc/rfc2818.txt.

[49] *ZIP Archive format* ftp://ftp.uu.net/pub/archiving/zip/doc/appnote-970311-iz.zip

[50] *RFC 2396 - Uniform Resource Identifier (URI)* http://www.ietf.org/rfc/rfc2396.txt

[51] *MIME Types*
http://www.ietf.org/rfc/rfc2046.txt and http://www.iana.org/assignments/media-types

[52] *3DES*
W/ Tuchman, "Hellman Presents No Shortcut Solution to DES," IEEE Spectrum, v. 16, n. 7 July 1979, pp40-41.

[53] *RFC 1423 Part III: Algorithms, Modes, and Identifiers*
http://www.ietf.org/rfc/rfc1423.txt

[54] *PKCS 5*
ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2

[55] *Java Cryptography API (part of Java 1.4)*
http://java.sun.com/products/jce/index-14.html

[56] *SHA-1*
U.S. Government, Proposed Federal Information Processing Standard for Secure Hash Standard, January 1992

[57] *Transport Layer Security*
http://www.ietf.org/rfc/rfc2246.txt, January 1999, The TLS Protocol Version 1.0, T. Dierks & C. Allen.

End Of Document