Subsystems

Draft

76 Pages

Abstract

This RFC proposes a design for OSGi Subsystems, where a subsystems is a collection of bundles with sharing and isolation semantics. The requirements for this RFC were defined and agreed in RFP 121.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 10.1.

`Source code is shown in this typeface.`

0.3 Revision History

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

Revision	Date	Comments
Initial	*11 19 2009*	*Created from RFP 121* *Graham Charters, IBM charters@uk.ibm.com*
0.1	*08 01 2010*	*Added Subsystem Definition Headers* *Graham Charters, IBM charters@uk.ibm.com*
0.2	*23 07 2010*	*Added JavaDoc. Added architecture overview/entities. Added subsystem types overview. Started adding deployment manifest details. Added archive definitions. Added lots of TODOs!* *Graham Charters, IBM charters@uk.ibm.com*
0.3	*01 09 2010*	*Consideration of XML or Java Properties files in Alternatives.* *Artefacts now in OSGI-INF instead of META-INF.* *ResourceProcessor uses new Coordination Service.* *SubsystemAdmin uses Futures for long-running activities.* *Use of EventAdmin instead of SubsystemListeners.* *Some details on Subsystem life-cycle (incomplete)* *Graham Charters, IBM charters@uk.ibm.com*
0.4	*22 11 2010*	*Graham Charters, IBM charters@uk.ibm.com* *Updated life-cycle to include transitional states.* *Removed use of Futures (transitional states mean they're of limited use).* *Added ability to snoop on internal subsystem events.* *Clarified resolution limitation (no cycles).* *Updated resource processors api to align with subsystem life-cycle operations.* *Updated references to RFC 138 to cover framework hooks.*
0.5	*01 02 2011*	*Graham Charters, IBM charters@uk.ibm.com* *Changed resource processor operations to sychronous.*

Revision	Date	Comments
0.6	*7th April 2011*	*Graham Charters, IBM charters@uk.ibm.com* *Updates based on Tom Watson's comments and discussion at Austin face-to-face, e.g.:* • remove SubsystemAdmin, moving capabilities to Subsytem. • new ResourceOperation approach for ResourceProcessors.
0.7	*06/06/11*	*John Ross, IBM, jwross@us.ibm.com* • Removed references to Resource Processors, which are no longer in scope. • Added cancel() method to Subsystem interface. Also added several missing Subsystem.State constants. • Added missing SubsystemConstants.EventType.UNINSTALLING constant. • Removed references to SubsystemEventConstants. These constants are now a part of SubsystemConstants. • Made a few comments. Responded to a few comments.
0.8	*12th August 2011*	*Graham Charters, IBM charters@uk.ibm.com* • Collapsed headers into single Subsystem-* set. • Collapsed extensions to single .ssa extension. • Filled out deployment manifest definition • Added description of transitive dependency provisioning with concept of root subsystem provision-policy. • Added description on handling share resource states.

Revision	Date	Comments
0.9	9/12/11	*John Ross, IBM, jwross@us.ibm.com* • *Accepted changes from revision 0.8 in SVN.* • *Merged changes from 0.8 branch not in SVN.* • *Several minor fixes.* • *Added default header value information, including the subsystem URI and file naming conventions.* • *Removed references to Coordinator and Coordination.* • *Updated state table.* • *Added reference to Bundle Collision Hook and bsnversion=managed.* • *Added start-order directive to Subsystem-Content and Deployed-Content headers.* • *Added use of Repository services requirement.* • *Addressed typo's, etc., highlighted in Glyn Normington's email on 8/25/11.* • *Updated Javadoc.*

1 Introduction

The OSGi platform provides a very appealing deployment platform for a variety of applications. In many scenarios, and especially enterprises, an application can consist of a large number of bundles. Administrators typically think in terms of applications for tasks such as deployment, configuration and update, rather than individual bundles, and therefore there is a mismatch in concepts between those provided by the OSGi framework and those familiar to an administrator.

The problem domain was introduced and requirements agreed in RFP 121. This RFC proposes a solution to address those requirements. Input to the design in this RFC will be drawn from a number of existing sources:

- SpringSource dm Server - *applications*
- Apache Felix Karaf – *features*

- Apache Aries – *applications*
- Newton – *systems*
- Eclipse – *features/installable units*
- DeploymentAdmin – *deployment package*

2 Application Domain

When System Administrators, Deployers and other people think about software they typically think in terms of systems performing a certain business function. A System Administrator's job might be to make sure that these systems stay up and running. The sysadmin doesn't have intimate knowledge of how these systems are constructed, but (s)he does have knowledge of the applications that compose the systems. There might be an ordering application, a payment application and a stock checking application. The sysadmin knows that these applications exist and that they must be monitored to ensure that they are in a healthy state.

Other people may think of a system yet on a different level. The CIO of the company may have a higher level view. He sees the system described above as a single application unit: the company's web shop and views all of the applications that together run in his organization as 'The System'.

On the other end of the spectrum, a developer might be tasked with~~by~~ implementing the payment system. This system might in turn be built up of other applications. There might be a currency conversion application, a credit card charge application and an electronic bank transactions application.

Bottom line is that what one person sees as an application~~,~~ might be a mere component to the next person in the chain. Application is a nestable concept and as such it is referred to by the more general word ‘Subsystem’. What the people described above have in common is that a pure OSGi bundle is typically not what they view as an application. An OSGi bundle is a component used to build applications, but applications themselves, no matter from what level you look at it, are always bigger than a bundle.

So from a conceptual point of view, applications are nestable. However, this does not mean that the runtime realization of these nested applications needs to exactly follow the nested structure. In OSGi systems, applications are built up from OSGi bundles, and although it would be desirable to view an OSGi system in a way that makes sense to the person looking at it, this does not mean that the bundles themselves need to be nested as well. The application view could be a virtual, organizational layer over a potentially flat bag of bundles that realize the applications at runtime. In some cases layering might be needed to enforce separation but in other systems the layering might not be used.

An application level view over an OSGi system might look like Figure 1.

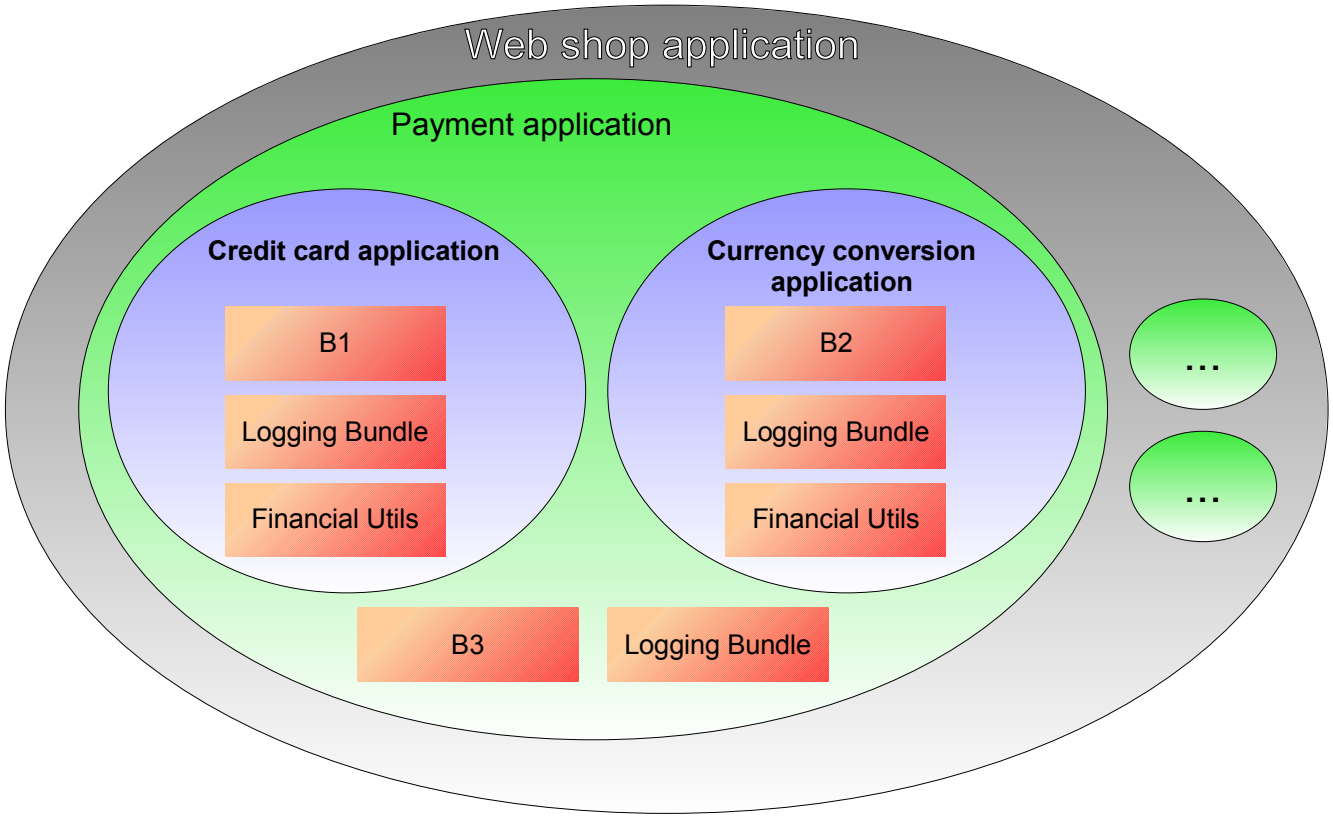

Figure 1: Application level logical view of a deployment

At runtime, the bundles composing this application might be deployed as a flat structure where the 'Logging Bundle' and 'Financial Utils' bundles are shared. This way of deploying the application is very efficient with regard to memory footprint. See Figure 2.

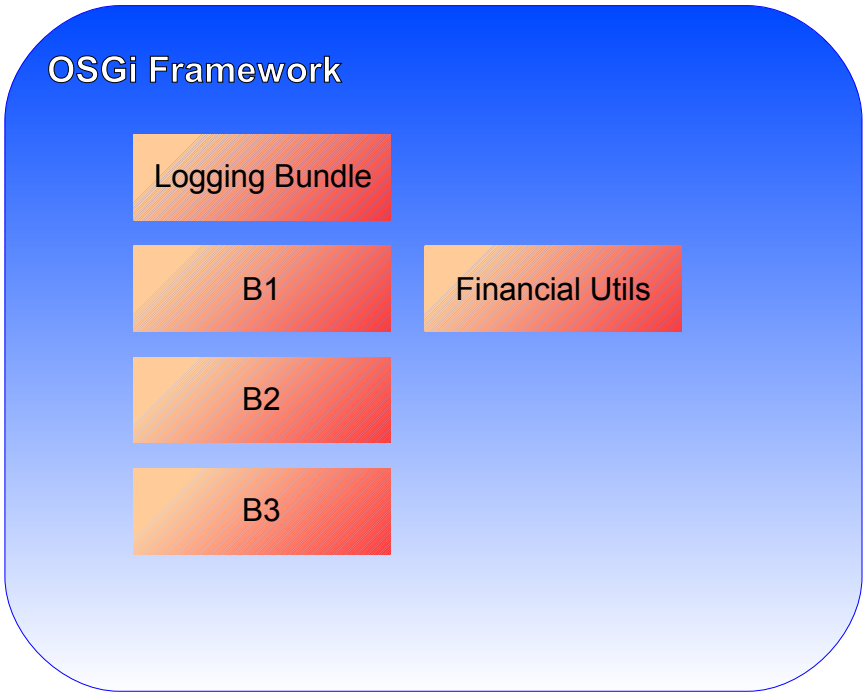

Figure 2: Runtime realization with sharing

Framework resolver hooks defined by RFC 138 could be used to isolate certain sub-systems while still sharing other bundles, like the logging bundle. This provides another runtime deployment option for the same system, providing more isolation. See Figure 3.

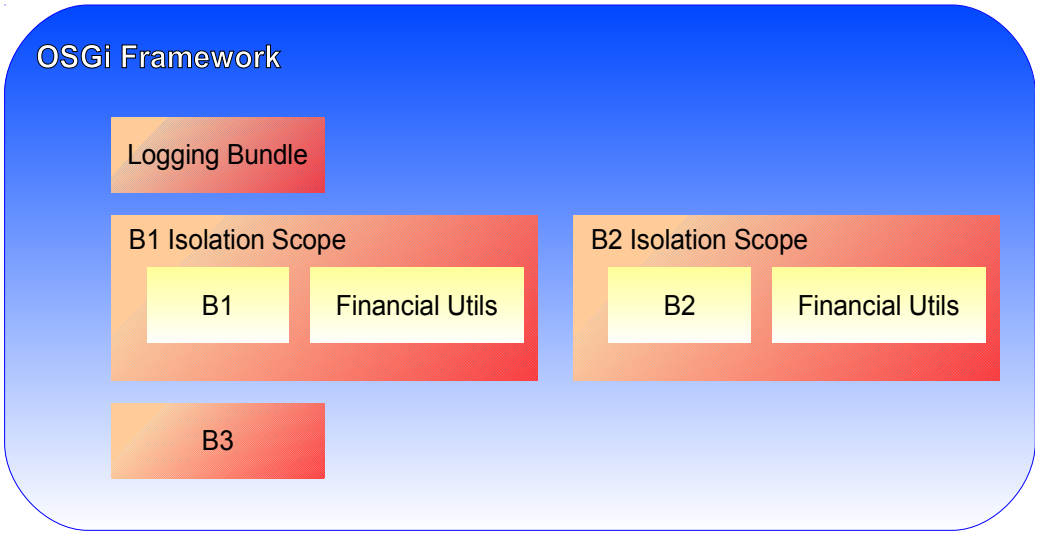

Figure 3: Runtime realization with less sharing and more isolation

Besides the applications installed in an OSGi framework, there are also bundles installed that are considered part of the OSGi framework infrastructure, but at runtime these bundles could become part of the application. Consider a bundle providing the logging service or a bundle providing the HTTP Service. OSGi Services are a great architecture for loose coupling, fostering sharing and re-use.

2.1 Related OSGi Specifications

2.1.1 OBR (RFC 112)

RFC 112 describes the OSGi Bundle Repository. This repository could provide an excellent place to store all the bundles required to deploy an application. It would therefore be important that any new designs resulting from this requirements document fully integrate and leverage the OBR.

However, using an OBR may not be right for every use-case so it should be optional. In some contexts development might start out relatively loose, using artifact repositories such as an OBR. Over time during the development process the dependencies in the project might harden and ultimately an OBR may not be appropriate any more. In a completely hardened situation dependencies might be obtained from a local directory created by an installer process or possibly a fixed corporate web site.

2.1.2 Framework~~s~~ Hooks (RFC 138)

Framework~~s~~ Hooks as proposed in RFC 138 provide a way of isolating or grouping subsystems within a single OSGi Framework. Hook configuration could be used to model an application at runtime, but there may also be implementations that do not require isolation so use of Framework~~s~~ Hooks should be optional.

2.1.3 Initial Provisioning

The Initial Provisioning spec does not relate to an application concept at all. It rather concerns ~~about~~ how to deploy initial bundles on a device. This specification is unrelated to this RF C~~P~~, but mentioned here to confirm that there is no potential overlap.

2.1.4 Deployment Admin

The Deployment Admin specification is about defining a file format to ship and deploy applications. Such an application is built up of multiple bundles and additional resources. Patching of applications, is also supported by this specification.

An additional feature provided by Deployment Admin, and which is considered useful for subsystems is the capability to roll back a deployment.

A major drawback of the Deployment Admin specification is that it explicitly prohibits sharing of bundles, which, besides preventing memory efficiency, could cause real problems when two applications use the same third-party bundle, as installing the same bundle twice is not allowed. Even different versions of the same bundle can not be installed simultaneously with Deployment Admin.

2.1.5 Application Admin

Application Admin provides the concept of an Application in OSGi. This specification could potentially be the basis of a runtime API into the Application Metadata and life-cycle system. It would most likely have to be extended to support all the use cases. Missing in the Application Admin spec is the Application Metadata, which should be declarative and available both inside the running OSGi framework as well as outside it to support OSGi tooling for subsystems.

Concepts missing from Application Admin:

- No file format to describe an application, purely API based
- No impact analysis for doing upgrades
- No interaction with OBR
- No application fingerprinting
- No runtime application extent analysis (what bundles beyond the core app set are used by this app)

2.1.6 Bundle Collision Hook (RFC 174)

The purpose of this hook is to provide a way to tell the framework what it should do when encountering a bundle with the same bsn and version. This new hook was necessary because the existing Find Hook could not be used during a bundle update because there is no valid bundle context at that time. The hook implementation will need to filter out any candidate having the same bsn and version as another bundle within the targeted region.

2.2 Terminology + Abbreviations

- ~~***Root Scope** – the scope at which the Subsystem Implementation bundle is installed. This is the top-most level at which the Subsystem service is available and therefore has no parent subsystem.*~~
- *Root Subsystem – The top-most subsystem in the hierarchy and, therefore, has no parent. It is always available as the starting point for installing other subsystems and resides at the same level into which the subsystems implementation bundle was installed.*
- *Resource – A bundle, subsystem, or configuration.*
- *Content Resource – A resource that is part of the subsystem content.*
- ***Transitive Dependencyies/Resources*** *– A rResources ~~(e.g. Bundles or Configuration)~~ that is not ~~explicitly~~ part of the subsystem content ~~of a subsystem~~ but is necessary to satisfy content requirements (e.g., package, bundle, or service dependencies) ~~from the subsystem contents that are not satisfied by the contents of the subsystem~~.*
- ~~***Unscoped Subsystem** – a subsystem that does not provide any package and/or service isolation. It effectively defines a set of bundles and or configuration that are provisioned into the OSGi framework in a flat bundle space.~~
- Unisolated Subsystem – A subsystem that does not restrict imported requirements or exported capabilities. It effectively defines a set of resources that are provisioned into the OSGi framework in a flat bundle space.
- ~~***Scoped Subsystem** – a subsystem that provides package and/or service isolation. The contents of the scoped subsystem will isolated by RFC 138 hook configuration.~~
- Isolated Subsystem – A subsystem that restricts imported requirements and exported capabilities. The contents are isolated using an RFC 138 hook configuration.
- ~~***Implicitly Scoped Subsystem** – a subsystem that provides implicit package, service and bundle isolation. Sharing is not explicitly declared. Packages, services and bundles are not shared outside the*~~

~~subsystem. Packages, services and bundle dependencies that are not provided inside the subsystem are share into the subsystem from outside.~~

- Implicitly Isolated Subsystem – A subsystem that implicitly restricts imported requirements and exported capabilities according to its type. For example, an application does not export any capabilities but will import any requirements not satisfied by the application's content.

- ~~***Explicitly Scoped Subsystem*** – a subsystems that provides explicit package, service and bundle sharing. Packages, service and bundles are not shared into, or out of, a subsystem unless explicit done so through hook configuration (see RFC 138).~~

- Explicitly Isolated Subsystem – A subsystem that explicitly restricts imported requirements and exported capabilities. The isolation policy is defined by the user within the subsystem manifest.

3 Problem Description

In today's OSGi framework, what is deployed is typically just a large set of bundles. To a person not familiar with the details of the design of these bundles it is often unclear what function these bundles perform and how they are related.

Some bundles might provide shared infrastructure (e.g. a bundle providing the OSGi Log service), while other bundles might together provide an application function (e.g. a set of bundles together implementing a web-based shopping application). Today it is not possible to find out from a high level what applications are installed in the OSGi framework. The only information available is the list of bundles.

The OSGi framework needs to be enhanced with a mechanism that makes it possible to declare applications. An application has a name meaningful to the deployer. It is typically composed of a number of key bundles and services, plus their dependencies. When the deployer requests the list of installed applications, he will get a manageable result that is typically much shorter than the list of installed bundles.

The deployer also needs to be able to perform actions on the application level. Installing, uninstalling, starting, and stopping should be possible on the level of an application.

A developer may wish to think in terms of an application consisting of a set of bundles and may wish to declare those bundles as belonging to the application.

3.1 Problem Scope

Graphical tools are out of scope for this RFC.

In scope would be the metadata needed to define the applications plus an API that would enable actions at the application level.

4 Requirements

The following requirements are taken from RFP 121, “Subsystem metadata and Lifecycle”. The requirement numbers used here are identical to those in RFP 121. Some requirements were ommitted from the final RFP and hence the numbering is intentionally not contiguous.

4.1 Subsystem Modeling

REQ 1. The solution MUST provide a means to describe a subsystem which can be accessed both inside a running OSGi framework (e.g. through an API) as well as outside of a running framework (e.g in a file).

REQ 2. The solution MUST define a subsystem definition format.

REQ 3. It MUST be possible to define a subsystem consisting~~in terms~~ of OSGi bundles.

REQ 4. It MUST be possible to make artifacts other than OSGi bundles part of a subsystem definition.

REQ 5. It MUST be possible to include a subsystem definition in another subsystem definition.

REQ 6. It MUST be possible to reference another subsystem in a subsystem definition, which makes the other subsystems a dependency.

REQ 7. It MUST be simple to define a scoped subsystem of the type defined in requirement 8 but with a default sharing policy that hides everything in the given subsystem from the parent of the given subsystem and makes all bundles (for wiring purposes only), packages and services visible inside the given subsystem from its parent.

REQ 8. It MUST be possible to scope subsystems. Scoped subsystems form a hierarchy. The subsystems, bundles, packages, and services belonging to a given scoped subsystem are visible within the scope of the given subsystem. A subsystem MUST have control over a sharing policy to selectively make packages and services visible to the parent of that subsystem. A subsystem MUST have control over a sharing policy to selectively make bundles (for wiring purposes only), packages and services visible inside that subsystem from its parent. Subsystems, bundles, packages, and services belonging to unscoped subsystems are always available to all peer subsystems (scoped or unscoped) and bundles.

REQ 9. Subsystems MUST be uniquely identifiable and versioned.

REQ 10. Subsystem definitions MUST be extensible. This will allow tools to store associated data along side the subsystem definitions.

4.2 Dependency Management

REQ 13. It MUST be possible to use constraints to declaratively identify bundles and other artifacts in a subsystem definition.

REQ 15. It MUST be possible to define a subsystem as a number of key bundles. The transitive dependencies of the subsystem are inferred from the meta-data of the key bundles.

4.3 Administration

REQ 18. The solution MUST provide an API to query the subsystems available in the OSGi container, their associated state (as a snapshot) and dependencies.

REQ 19. The fidelity of the subsystem states MUST be sufficient to capture the possible states of its constituents

REQ 20. The solution MUST provide an API to drive the subsystem life-cycle.

REQ 22. The solution SHOULD provide an API to do impact analysis regarding replacing one or more bundles or subsystems. This impact analysis should list the subsystems affected and describe how these subsystems are using the affected bundle: either by importing an interface or a class from it, purely through services or through the extender pattern. It should also describe existing transitive dependencies and new transitive dependencies.

REQ 24. The APIs that modify the system SHOULD not leave the system in an inconsistent state. .

4.4 Runtime

REQ 26. The solution MUST allow a single bundle to be part of multiple subsystems.

REQ 27. The solution MUST allow multiple versions of the same bundle.

REQ 28. Subsystems MUST have a well-defined life-cycle.

4.5 Development Support

REQ 29. The solution MUST support the development process by allowing bundles to be updated with other bundles that have the same version as the previous, similar to how Maven supports SNAPSHOT versions of artifacts.

4.6 RFC 138 Compatibility

REQ 30. The solution MUST be compatible with the scoping mechanisms possible through RFC 138.

5 Technical Solution

Subsystems enable a number of resources to be managed as a single entity, such as a set of bundles with associated configuration. Subsystems also allow selective sharing of capabilities and requirements (borrowing the terms from rfc 154), such as packages and services. For example, a subsystem might be defined in terms of a set of bundles that share a number of packages and services between themselves, but only share a subset of

those capabilities outside the subsystem. This selective sharing enables the creation of assets that are more coarse-grained than individual bundles, and exhibit their own modularity.

Subsystems simplifies the task for deploying the collection or resources by integrating the OSGi Bundle Repository (OBR) technology to aid resolution of the collection and defining a mechanism for describing and processing the resources to be deployed, including content and transitive dependency resources. Subsystems also describes how the life-cycle of the content and transitive resources are affected by the life-cycle of a subsystem.

Subsystems are designed to be easy to create for those familiar with bundle development. They therefore have many concepts and design choices in common with bundles. For example, a manifest format is used to define a subsystem, as well as describe its deployment, and they have a symbolic name and version for their identity.

The subsystem service design consists of the following elements:

1. Subsystem metadata for defining subsystems.

2. An API for the management of subsystems; install, uninstall, updatestart, stop, event notification.

5.1 Subsystems Architecture

The subsystems architecture is shown in figure 4.

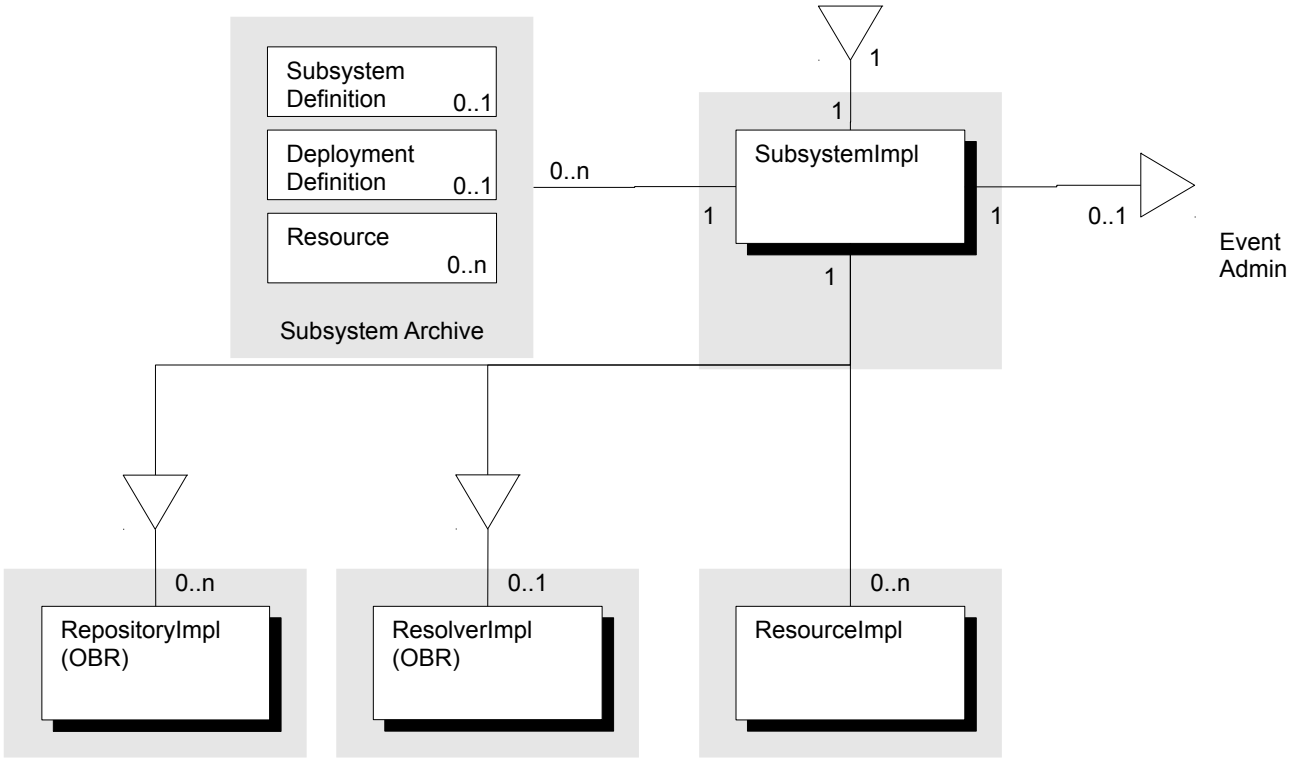

Figure 4: Subsystem Architecture

The main entities in the architecture are as follows:

- *Subsystem Definition* – A Subsystem Definition is a description of a subsystem that is processed by the Subsystem Admin service. For example, it describes the contents of the subsystem and, depending on the type of subsystem, it also describes any packages and services that are shared in our out. The subsystem definition is optional.

- *Deployment Definition* – A Deployment Definition describes the exact resources to provision for subsystem (i.e. the resolved content and any resources require to satisfy the subsystems dependencies), and any sharing poilicies (i.e. the packages and services that can be shared in or out of the subsystem). The deployment definition is optional.

- *Resource* – A Resource is any artifact, such as a bundle or configuration, that may be required when provisioning a subsystem.

- *Subsystem Archive* – An archive used to package a subsystem for installation into an OSGi runtime. The archive optionally contains a subsystem definition, a deployment definition, and resources.

- *Subsystem* – An implementation of this specification that provides the ability to install, uninstall and locate subsystems.

- *Repository* – Subsystems are not required to be, and typically won't be, transitively closed and therefore some amount of resolution will be required during installation to ensure all requirements are satisfied. Zero or more Repositories may be required in order to fully provision a Subsystem, including its transitive dependencies. Subsystems defines the use of repository services defined by the OBR specification, RFC 112. If OBR repositories are not available then a subsystem implementation is free to use others. If repository services are available, subsystems implementations must query them in service ranking order and include any found capabilities for possible use by a resolver. Implementations are free to give higher priority to capabilities found by other means.

- Resolver – A Resolver is used by the Subsystem implementation to determine whether the subsystem resolves (i.e. is transitively closed) or requires additional resources in order to satisfy all its requirements. The Resolver is defined by the OBR specification, RFC 112.

5.2 Subsystem Types

Three types of subsystems are defined in this specification; Application, Composite Bundle and Feature. Each differs in the way in which they share requirements and capabilities and are described in more detail in the next few sections.

5.2.1 Application Subsystems

An application subsystem is a subsystem with a sharing policy associated with what people would often consider to be an application. An application subsystem consists of a number of bundles and other supporting resources. The content bundles share package and service dependencies with each other. An application does not share any package or service capabilities to bundles outside the application. Any package or service requirements that are not satisfied by the content bundles themselves are automatically imported from outside the application.

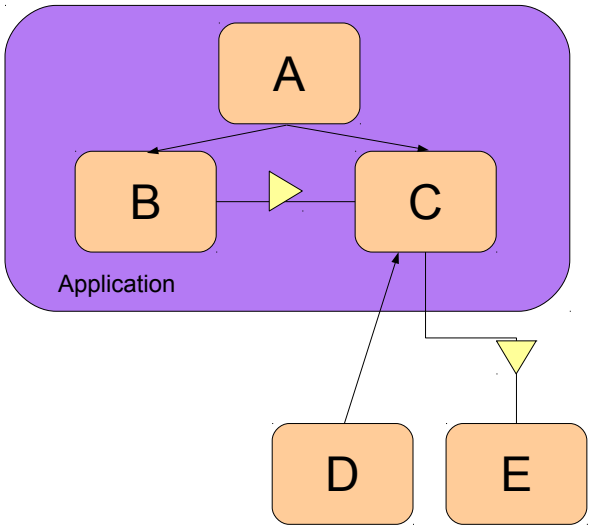

Figure 5: Application package and service sharing example.

An example of this point is shown in figure 5. An application consisting of 3 content bundles, A, B and C, is shown. Bundle A is exporting an API package to bundles B and C. Bundle C is providing a service which is being used by Bundle B. Bundle C also requires a package and service, neither of which are provided by bundles inside the application. These are therefore automatically shared into the application and in this example provided by bundles D and E, respectively.

5.2.2 Composite ~~Bundle~~ Subsystems

A composite subsystem is a subsystem with a fully explicit sharing policy. A composite consists of a set of content bundles and other supporting resources. These content bundles share package and service dependencies inside the composite. By default a composite does not share packages or services into or out of itself. Any packages or services that are to be shared into or out of the composite, must be explicitly identified in the composite subsystem definition.

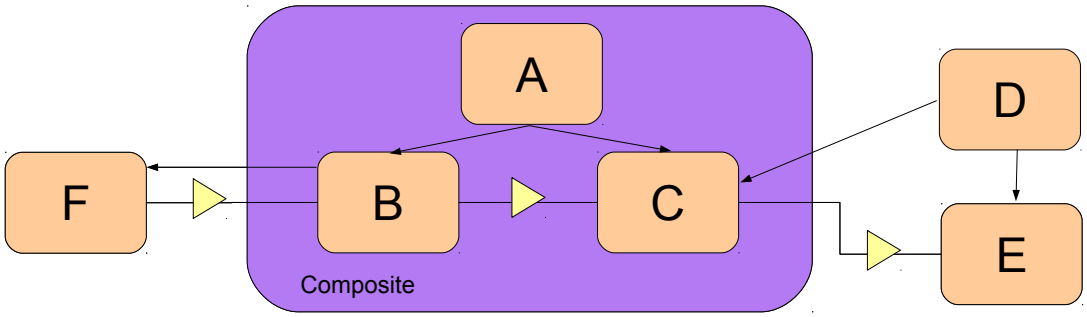

Figure 6: Composite package and service sharing example.

Figure 6 shows an example composite subsystem that is sharing packages and service both in and out of the composite for use by bundles inside and outside the composite, respectively. In this example, bundles A, B and C have various internal package and service dependencies. However, the composite also exports a package and service provided by bundle B that are being used by bundle F. In addition to this, the composite is importing a package provided by bundle D and a service provided by bundle E. Both of these are then being used by content bundle C.

It is worth noting that a consequence of the resolution process outlined in section 5.2.7 is that a composite subsystem is not permitted to be involved in a package dependency cycle with other peer subsystems or bundles. Such cycles will result in the subsystem failing to resolve.

5.2.3 Feature Subsystems

A feature subsystem is a subsystem that does not impose any isolation. A feature consists of a set of bundles and associated resources. The packages and services provided by the content bundles are all automatically made available to bundles outside the feature. Packages and services required by the content bundles can automatically be satisfied by bundles outside the feature. The content bundles can be required by bundles outside the feature and the content bundles can required by bundles from outside the feature. The main purpose of a feature subsystem is to enable the life-cycle management of a set of bundles.

Subsystem

An implementation of this specification provides a subsystem service for managing the life-cycle of subsystems. The service implements the org.osgi.service.subsystem.Subsystem interface (see section 6). Each subsystem service instance manages subsystems directly visibile to it in the subsystem hierarchy, as illustrated in figure 7.

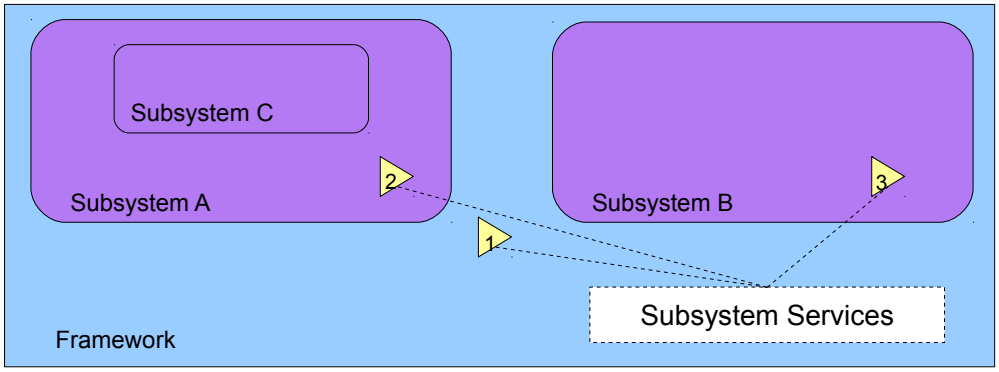

Figure 7: Subsystem Service instance model.

Figure 7 shows four Subsystem service instances registered, each numbered to aid description. In this example, at the outer-most level (the *root scope*), Subsystem service 1 is managing Subsystems A and B. A and B and be retrieved using **Subsystem.getChildren()**. Because this service is in the *root scope* a call to **Subsystem.getParent()** would return `null` as it has no parent subsystem. At the next level down, Subsystem service 2 is managing subsystem C and Subsystem service 3 is not managing any subsystems. A call to **Subsystem.getChildren()** on service 2 would return Subsystem C and **Subsystem.getParent()** would return Subsystem A. A call to **Subsystem.getChildren()** on service 3 would return and empty subsystem Collection as it has no child subsystems.

A subsystem implementation may choose to hide the subsystem service to prevent a subsystem's content bundles from creating their own subsystems. This is illustrated in Subsystem C, which does not contain a subsystem service.

Note, the Subsystem service view described above is logically what is expected, however a most likely implementation strategy would be to have a single service factory registration which is visible inside each subsystem and returns the appropriate subsystem instance based on the subsystem in which the requesting bundle lives.

5.2.4 Life-cycle

Subsystems have a life-cycle similar to that of Bundles. Operations on the Subsystem (e.g. install, uninstall, start and stop) cause it to move through its life-cycle. The operations also cause equivalent operations on the Subsystem content (e.g. start on a Subsystem will cause start on any content bundles). In these circumstances, the processing of the Subsystem content must also respect any life-cycle policy of the content resources. For example, starting a subsystem that contains bundle fragment resources must not require those fragments to start as this would be counter to the life-cycle policy of bundle fragments.

The use and behavior of subsystem life-cycle in the presence of the start level services is out of scope.

All subsystem states are defined by the `Subsystem.State` enum, for example, `Subsystem.State.INSTALLED`. For ease of reading these will hereafter be referred to by their short name, e.g. `INSTALLED`.

The life-cycle state of a subsystem is a reflection of the states of the contents and the last action performed through the subsystem api. For example, a Subsystem only transitions to `ACTIVE` when all its contents have successfully started, triggered by calling **start** on the subsystem. However, if a program uses the reflective api to change the state of an individual content bundle, this will not result in a change of state for of the subsystem. A

subsystem's state also reflects the cases where only a subset of it's contents have reached some target state. These states are referred to as 'transitional states'. Examples include `INSTALLING` and `RESOLVING`.

A subsystem's state is persistent, however, the state of their contents is not. Restarting a framework will cause subsystems to attempt to reach their previous states, which will try to bring the state of the content resources to the same state. For example, if a subsystem was `ACTIVE` when the framework was stopped, then when that framework is started again, the subsystem will try to reach the `ACTIVE` state again, which will result in an attempt to **start** all content resource, irrespective of their actual state when the framework was stopped.

Figure 8 shows the life-cycle for a Subsystem.

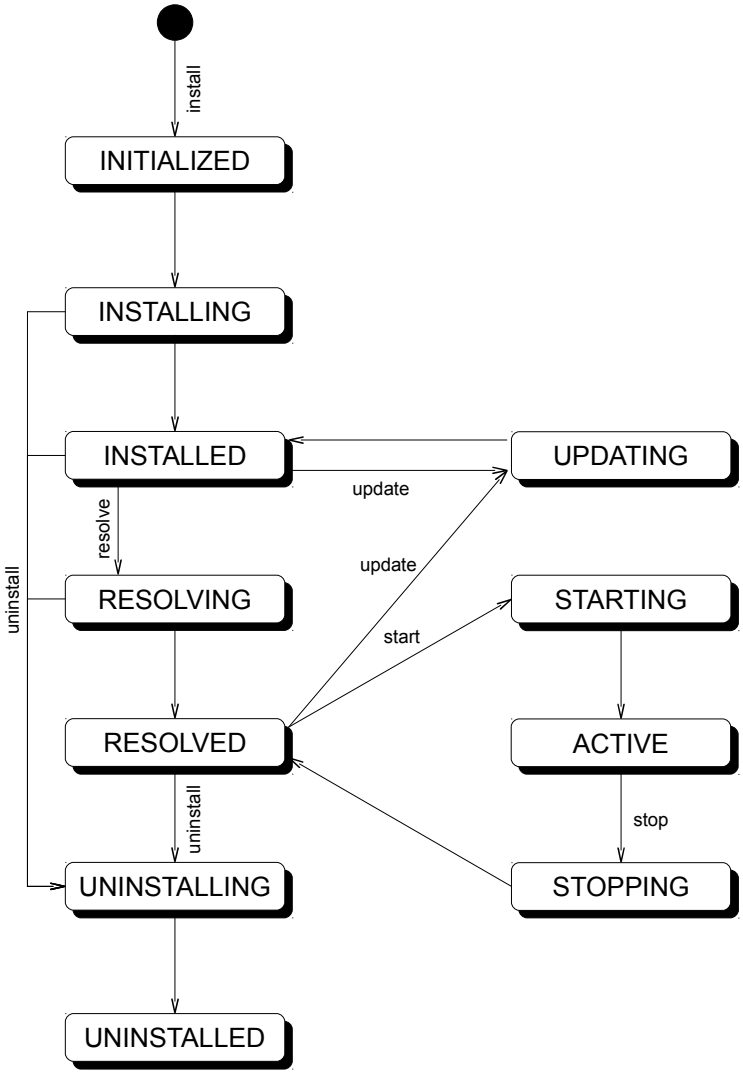

Figure 8: Subsystem Lifecycle

5.2.4.1 Events

When the EventAdmin service is available in the root scope, The subsystem runtime will produce events based on the Subsystem life-cycle. All events are delivered asynchronously. The Event Topic of these events is:

```
org/osgi/service/Subsystem/<event-type>
```

Whenever an event type is mentioned in this document, it should be assumed it belongs to the above topic, unless stated otherwise.

The following event types are generated based on the life-cycle of the Subsystem

Event Type	**Description**
INSTALLING	Indicates a subsystem has started installing
INSTALLED	Indicates a subsystem has completed installed
RESOLVING	Indicates a subsystem has started resolving.
RESOLVED	Indicates a subsystem has been resolved
STARTING	Indicates a subsystem has begun starting.
STARTED	Indicates a subsystem has become active.
STOPPING	Indicates a subsystem has begun stopping

STOPPED	Indicates a subsystem has been stopped
UPDATING	Indicates a subsystem has started updating
UPDATED	Indicates a subsystem has been updated
UNINSTALLING	Indicates a subsystem has started uninstalling.
UNINSTALLED	Indicates a subsystem has been uninstalled
CANCELING	Indicates that a subsystem operation is being canceled.
FAILED	Indicates a subsystem life-cycle operation (e.g. install, update) failed
CANCELED	Indicates an asynchronous subsystem operation was canceled (e.g. install or update).

Subsystem events contain the following properties to help identify the subsystem, the time at which the event occurred and any exceptions that may have occurred.

- `SUBSYSTEM_ID` – the id of the subsystem for which the event was being generated (from SubsystemConstants).
- `SUBSYSTEM_LOCATION`. - the location of the subsystem for which the event was generated (from SubsystemConstants).
- `SUBSYSTEM_SYMBOLICNAME` – the symbolic name of the subsystem from which the event was generated (from SubsystemConstants)
- `SUBSYSTEM_VERSION` – the symbolic name of the subsystem from which the event was generated (from SubsystemConstants)
- `TIMESTAMP` – the time at which the event was generated (from EventConstants).
- `SUBSYSTEM_STATE` – the current state of the subsystem (e.g. CANCELING) (from SubsystemConstants).
- `EXCEPTION` – contains any exception that caused the event (from EventConstants).
- `EXCEPTION_CLASS` – filled in as defined by event admin (from EventConstants).
- `EXCEPTION_MESSAGE` - filled in as defined by event admin (from EventConstants).

5.2.5 Listening for Internal Subsystem Events

Users of subsystems may need to be aware of the internal goings on of a subsystem. One use case for this is to be able to determine when a subsystem is 'ready'. Readiness may not necessarily directly correspond to one of the subsystem states outlined in section 5.2.4, For example, a library bundle might only need to reach the `RESOLVED` state, whereas a blueprint bundle might only be considered ready once its blueprint container service has been registered.

The subsystems runtime enables users to snoop on the inner workings of subsystems through EventAdmin. Events that occur inside a subsystem and which are not normally propagated outside the subsystem can be

observed by registering an event handler in the subsystem service's root scope. Internal events are re-published using the following event topic pattern:

```
org/osgi/service/SubsystemInternals/<original-topic>
```

The use of the topic token “SubsystemInternals” keeps internal and external events in separate peer topic spaces, thus allowing wildcarding.

<original-topic> is the original topic of the event being re-published. The event published is the event admin event that would have been generated internally. For example, a bundle `STARTED` event would be published as:

```
org/osgi/service/SubsystemInternals/osg/osgi/framework/BundleEvent/STARTED
```

Note, because event admin is not subsystem aware, it is necessary to install event admin in every level at which event admin is to be used.

The following properties are added to the event to identify the subsystem from which the event came. These properties allow handlers to filter for a specific subsystem:

- `SUBSYSTEM_ID` – the id of the subsystem from which the event was generated (from SubsystemEventConstants).
- `SUBSYSTEM_LOCATION` - the location of the subsystem from which the event was generated (from SubsystemEventConstants).
- `SUBSYSTEM_SYMBOLICNAME` – the symbolic name of the subsystem from which the event was generated (from SubsystemEventConstants)
- `SUBSYSTEM_VERSION` – the symbolic name of the subsystem from which the event was generated (from SubsystemEventConstants)
- `TIMESTAMP` – the time at which the event was generated (from EventConstants). This value should be set only if not already present.
- `SUBSYSTEM_STATE` – the current state of the subsystem. This is used when the state is not indicated by the event type (e.g. CANCELING) (from EventConstants).
-

5.2.6 Installing a Subsystem

Subsystems are installed using one of the service's **install** methods. These methods carry the same arguments as BundleContext.installBundle and the same semantics, such as a null InputStream meaning the InputStream is created from the location identifier. Subsystem install operations are potentially long-running and are therefore asynchronous. Each install method returns a Subsystem with the installation process initiated, but not necessarily complete.

The following steps are required to install a subsystem:

1. If there is an existing subsystem containing the same location identifier as the Subsystem to be installed then the existing Subsystem is returned.
2. If this is a new install, then a new `Subsystem` is created with its id set to the next available value (ascending order)

3. The subsystem's state is set to `INSTALLING` and if EventAdmin is available, an event of type `INSTALLING` is fired.
4. The following installation steps are then started and performed asynchronously and the new subsystem is returned to the caller:
5. The subsystem content is read from the input stream.
6. If the subsystem requires isolation (i.e. is an application or a composite), then isolation is set up while the install is in progress, such that none of the content bundles can be resolved. This isolation is not changed until the subsystem is explicitly requested to resolve (i.e. as a result of a **Subsystem.start()** operation).
7. If the subsystem does not include a deployment manifest (see 5.3.6), then the subsystem runtime must calculate one as described in section 5.3.6.17.
8. The resources identified in the deployment manifest are installed into the framework. All content resources are installed into the Subsystem, whereas transitive dependencies are installed into an ancestor subsystem as describe in section 5.2.6.1. If any resources fail to install, then failure is handled as described in 5.2.6.2. Transitive resources are free to resolve and start independent of the subsystem they were installed for.
9. The subsystem's state is set to `INSTALLED` and if EventAdmin is available an `INSTALLED` event is fired.

Note, if the subsystem requires isolation and it is in the `INSTALLING` or `INSTALLED` state, none of it's content bundles must be permitted to resolve.

5.2.6.1 Resource Installation

Two classes of resource are potentially installed during subsystem installation; content resources and transitive resources

Content Resources – these are resources identified by the Subsytem-Content (5.3.2.4) header of the subsystem manfiest. The deployment manifest Deployed-Content header (5.3.6.5) identifies the exact version of each content resource to be provisioned. Content resources are installed directly into the Subsystem.

Transitive Resources – these are the resources required to provide capabilities to satisfy requirements from the content resources that are not satisfied by capabilities provided by the content resources themselves or the capabilities provided by the target environment.

Transitive resources are provisioned into the ancestor hierarchy of the subsystem. The final destination of the transitive resources is determined by the subsystem or its ancestors. A subsystem states whether it is willing to hold transitive dependencies by specifying the “provision-policy:=acceptTransitive” directive on the Subsystem-Type header. Transitive dependencies are provisioned into the first subsystem subsystem with acceptTransitive encountered when traversing up the subsystem hierarchy, including the subsystem being provisioned. If no such subsystem is encountered, then the transitive dependencies are provisioned into the *root scope*.

Figure 9 shows a simple example of provisioning transitive dependencies. An application subsystem has been installed into a composite subsystem where the composite has opted to allow transitive dependencies. The application has three content bundles A, B and C, and two transitive dependency bundles, D and E. The two transitive dependencies have been provisioned into the parent composite subsystem.

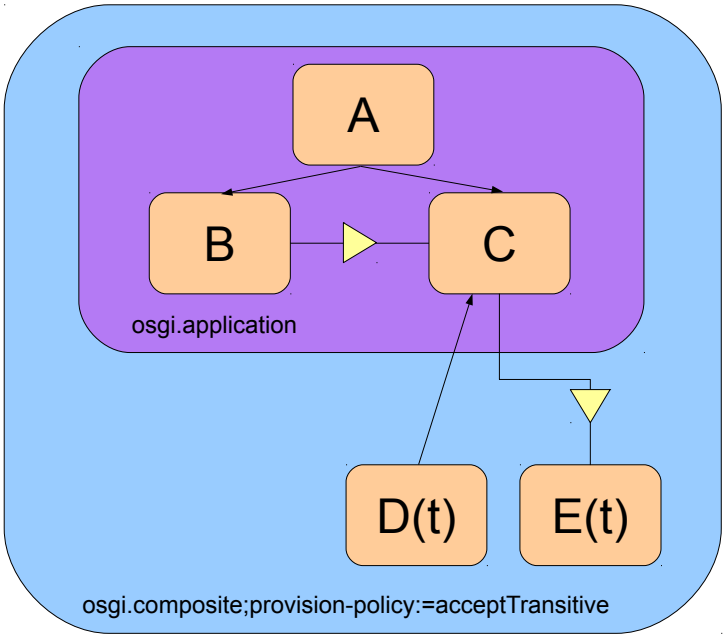

Figure 9: Simple transitive resource provisioning example

Figure 10 shows an example of the application from the previous example, but with the application opting in to hold transitive dependencies. Because the application itself is the first root encountered, the transitive bundles, D and E are installed into the application subsystem, rather than the parent.

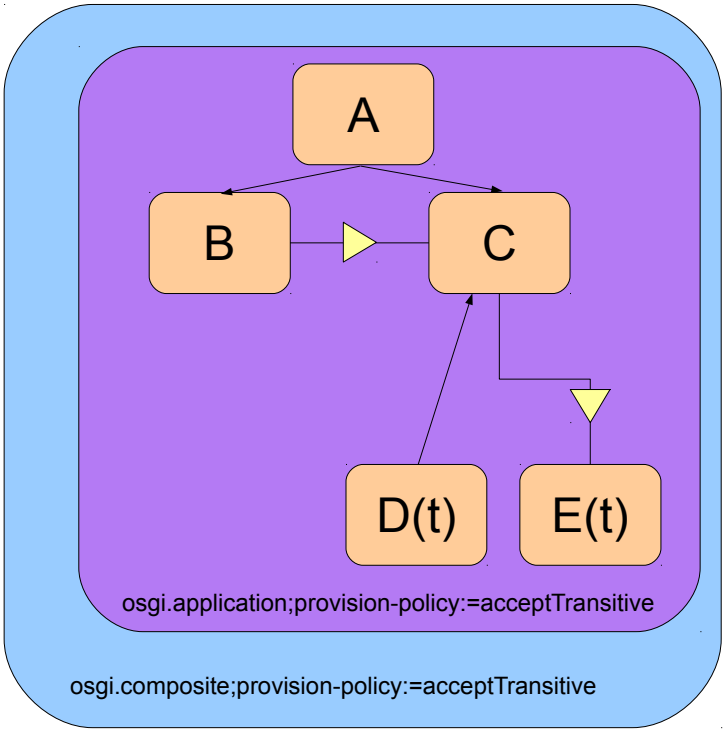

Figure 10: Transitive resource provisioning with multiple roots

Figure 11 shows an example where the isolation policy of one of the subsystems prevents the installed applications for accessing all the capabilities of its transitive dependencies. In this example, the first 'root' subsystem encounter is a two generations up from the application subsystem. This is where the transitive dependency bundles D and E are installed. However, the isolation imposed by an intermediate composite subsystem prevents the application from accessing the service provided by bundle E. This is considered a deployment error.

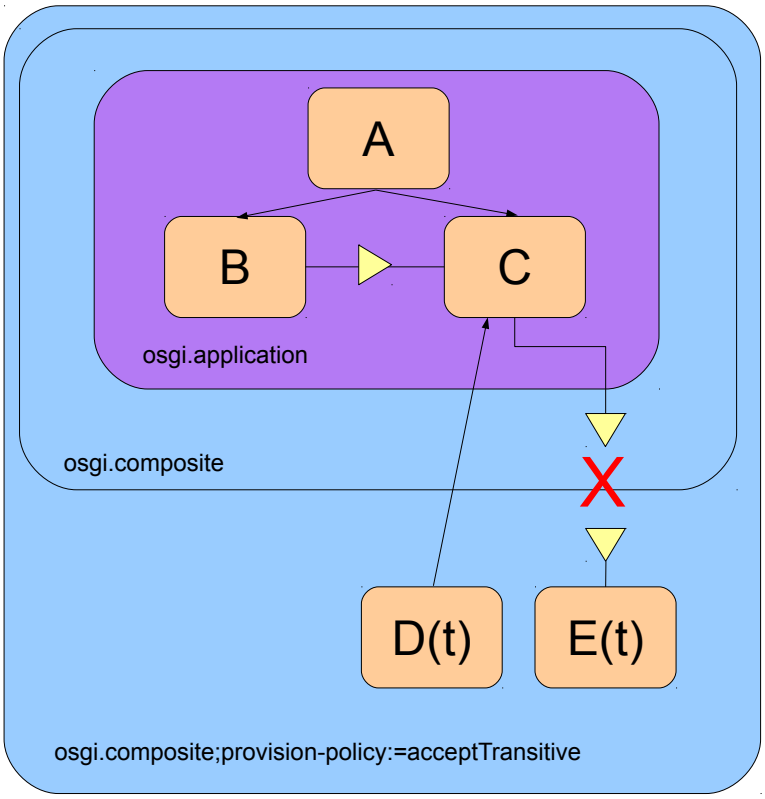

Figure 11: Transitive resource provisioning example with isolation error

5.2.6.2 Installation Failures

If the installation of a Subsystem fails, for example, due to problems opening an InputStream, then a subsystem FAILED event is fired. The exception that caused the failure is set in the event properties EXCEPTION, EXCEPTION_CLASS and EXCEPTION_MESSAGE. The subsystem id and location are also provided in the event to identify the failed subsystem, and finally, the timestamp is set to indicate when the failure occurred.

Failure to install one of the content resources (e.g. a content bundle) must result in a failure to install the subsystem and an attempt to undo any associated installation activities. The subsytem's state is then set to UNINSTALLED and the UNINSTALLED event fired via EventAdmin, if available.

5.2.6.3 bsnversion Property

The framework property bsnversion must be set to “managed” for subsystems. This will be the default value for the OSGi Core 4.4 release. Implementations will need to provide a Bundle Collision Hook as discussed in RFC 174.

5.2.7 Resolving a Subsystem

Resolution of a subsystem only happens as a result of an explicit request (i.e. by calling **Subsystem.start()**. Because start is an asynchronous operations there is no need for resolution to be done asynchronously. A subsystems state is only considered RESOLVED when all its content bundles are RESOLVED. The subsystem runtime issues a RESOLVED event when this condition is met.

To ensure a consistent and 'whole' view is presented for the composite subsystem externals, resolution happens in two steps. Resolution for Feature subsystems, which are transparent and Application subsystems, which do not export anything, can happen in a single step. Subsystem resolution occurs as follows:

1. Resolution is initiated with a call to **Subsystem.start()**.
2. If the subsystem is in the UNINSTALLING or UNINSTALLED state then a **SubsystemException** is thrown.
3. If the subsystem is already resolved, then no action is taken.
4. The subsystem's state is set to RESOLVING and a RESOLVING event is fired via EventAdmin, if available.
5. If the subsystem is one with isolation, then isolation is configured such that its content bundles are permitted to resolve against each other and any externals that it imports. If the subsystem is a composite then its exports are not yet made available for resolution.
6. An attempt is made to resolve the content bundles and any transitive dependencies that are not resolved (including subsystem dependencies).
7. If the subsystem fails to resolve (perhaps due to a content bundle not resolving) then the subsystem state is set to INSTALLED. A subsystem FAILED event is fired via EventAdmin, if available.
8. If the subsystem is a composite, then once all content bundles are RESOLVED, the subsystem makes any packages it exports available for resolution. Any subsequent resolution is then able to wire to packages provided by the composite subsystem. The subsystem's state is set to RESOLVED and a RESOLVED event is fired via EventAdmin, if available.

A consequence of this approach is that a composite subsystem is not permitted to be part of a package dependency cycle with other peer bundles or composite subsystems. If cycles exist then the composite subsystem will never resolve.

5.2.8 Starting a Subsystem

Subsystems are started by calling the Subsystem **start** method. Starting a subsystem performs the following steps:

1. If the Subsystem is in the `UNINSTALLING` or `UNINSTALLED` state, then a `SubsystemException` is thrown.

2. If the Subsystem is in the `ACTIVE` state, then the method returns immediately.

3. The starting process is initiated asynchronously and the method returns. From this point forward a framework restart must automatically start the subsystem (i.e. the subsystem's start state is persisted).

4. If there are existing content bundles that have not been resolved, then the runtime must attempt to resolve these first, as described in section 5.2.7. If the subsystem fails to resolve then the start does not continue.

5. Once all bundles are `RESOLVED` (i.e. the subsystem is resolved) then the subsystem's state is set to `STARTING` and if EventAdmin is available, an event of type `STARTING` is fired.

6. Resources are started in their start-level order irrespective of the framework start-level. The subsystem runtime is free to start bundles with the same start-level in any order it chooses. If a resource is already active (e.g. in the case of two intersecting feature subsystems), then it is skipped. Note, the subsystem content resources must only be transiently started, meaning the persistent start of the containing subsystem will cause them to be started at the appropriate time. This avoids circumstances where content resources could be started ahead of the subsystem being set up.

7. Failure to start one of the content resources (e.g. a content bundle) must result in a failure to start the subsystem and an attempt to undo any associated activities. The subsystem state must be set to `RESOLVED`. The `FAILED` event is fired with the subsystem identification information, timestamp, and any exception information regarding the reason set in the appropriate event properties.

8. If all content resources successfully start (note, for some resource types, start may be a no-op, for example, in the case of a bundle fragment) then the subsystem's state is set to `ACTIVE`, and the `STARTED` event is fired via EventAdmin, if available.

Note, as each content bundle is activated, it may register and consume services before the entire subsystem is activated. This behavior is inconsistent for composite subsystems, where the design goal is to present a “whole view”, however, to achieve this would require complex alternatives, such as replaying service events.

5.2.9 Stopping a Subsystem

Subsystems are stopped by calling the Subsystem **stop** method. Stopping a subsystem performs the following steps:

1. If the Subsystem is in the `UNINSTALLING` or `UNINSTALLED` state then an `IllegalStateException` is thrown.

2. If the subsystem's state is not `STARTING` or `ACTIVE` then this method returns immediately.

3. The Subsystem's state is set to `STOPPING` and the `STOPPING` is fired via EventAdmin, if available.

4. The stopping process is initiated asynchronously and the method returns. From this point on, the subsystem stop state is persisted. The stopping process performs the following actions:

5. Resources are stopped in their reverse start-level order. The subsystems runtime is free to stop bundles with the same start-level in any order it chooses.

6. Failure to stop any resource must result in a `FAILED` event being fired with the subsystem identification information, timestamp, and any exception information regarding the reason set in the appropriate event properties. Stop processing continues to try to stop any remaining resources.

7. Once an attempt has been made to stop all resources, the Subsytem's state is set to `RESOLVED` and the `STOPPED` event is fired via EventAdmin, if available.

5.2.10 Stopping the Framework

When the framework is stopped, all subsystems must also be stopped before the start level starts taking down the bundles. This ensures subsystem stopping is managed appropriately by the subsystem runtime and not as an accidental and undesirable side-effect of the framework stopping individual bundles.

5.2.11 Unstalling a Subsystem

Subsystems are uninstalled by calling the Subsystem **uninstall** method. Uninstalling a subsystem results in it being put into the `UNINSTALLED` state and sending an `UNINSTALLED` event. The Framework must remove any resources related to this subsystem that it is able to remove. If this subsystem has exported any packages, the Framework must continue to make these packages available to their importing bundles or subsystems until the PackageAdmin.refreshPackages method has been called or the Framework is relaunched. The following steps are required to uninstall a subsystem:

1. If this subsystem's state is UNINSTALLING or `UNINSTALLED` then this method returns immediately.
2. If this subsystem's state is `ACTIVE`, `STARTING` or `STOPPING`, this subsystem is stopped as described in section 5.2.9.
3. The Subsystem's state is set to `UNINSTALLING` and the `UNINSTALLING` event is fired via EventAdmin, if available.
4. The uninstall process is initiated asynchronously and the method returns. The uninstall process performs the following actions:

5. Resources are uninstalled.

6. Failure to uninstall any resource must result in a `FAILED` event being fired with the subsystem identification information, timestamp, and any exception information regarding the reason set in the appropriate event properties. Uninstall processing continues to try to uninstall any remaining resources.
7. Once an attempt has been made to uninstall all resources, the Subsytem state is set to `UNINSTALLED` and the `UNINSTALLED` event is fired via EventAdmin, if available.

5.2.12 Canceling a Subsystem operation

Asynchronous subsystem operations can be canceled through the ~~the~~ Subsystem **cancel** method. Cancellation of a subsystem performs the following steps:

1. If the subsystem is not in a transitional state initiated through the Subsystem service (i.e. not `INSTALLING`, `UNINSTALLING`, `STARTING`, `UPDATING`, `STOPPING`), then an `IllegalStateException` is thrown.

2. It is permissible for an implementation to delay canceling if it is in the final steps of completing an in-flight operation. This would result in the cancel operation blocking briefly and then an `IllegalStateException` being thrown.

3. A `CANCELING` event is fired via EventAdmin, if available.

4. The cancel process is initiated asynchronously and the method returns. The cancel process then performs the following actions:

5. The runtime attempts to undo all actions performed as part of the current operation. ~~This is done by failing the coordination used to coordinate any resource processors used which should result in reverting any actions taken under the coordination.~~

6. If the undo process fails for any resource, then a `FAILED` event is fired via EventAdmin, if available and the subsystems state remains in the transitional state.

7. If the undo process is successful, then the subsystem state is set to the state it was in prior to beginning the transition, and a `CANCELED` event is fired via EventAdmin, if available.

5.2.13 Retrieving Subsystems

Subsystems form a tree hierarchy. The child subsystems of a subsystem can be retrieved by calling the Subsystem **getChildren()** method. The parent subsystem of a subsystem can be retrieved by calling its **getParent()** method.

5.2.14 Shared Resource States

Resources can be shared between subsystems in two ways:

1. As common transitive resource dependencies between two sibling subsystems (see Figure 12).

2. As common content resources between two sibling feature subsystems (see Figure 13).

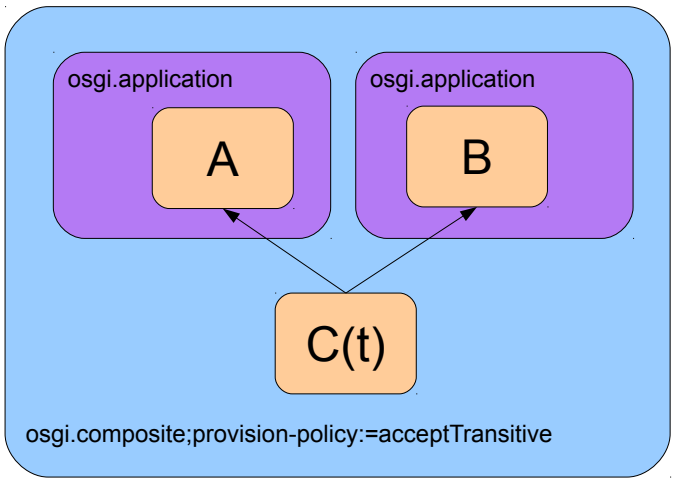

Figure 12: Two subsystems with a common transitive dependency

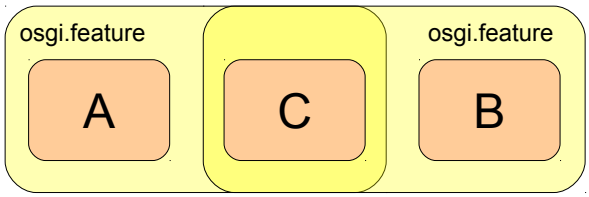

Figure 13: Two feature subsystems with shared common content

In both cases, the subsystems that cause the shared resource to be installed must share the responsibility for that resource's life-cycle. The shared resource life-cycle is managed such that it remains in the highest state required (see 5.2.14.1 for state precedence order) by each of the other resources with a dependency on that resource. This includes actions performed directly on the resource. For example, if the resource is installed and then is subsequently used as a transitive dependency, uninstalling the thing which requires the resource as a transitive dependency does not unisnstall the resource.

Direct resource operations override the states required by things that depend on the resource. For example, performing an uninstall on the resource will uninstall that resource, irrespective of whether it is still required as a transitive dependency of another subsystem. This allow admin agents to perform updates of shared resources.

It's worth observing that when all things with requirements on transitive dependencies are uninstalled, their transitive dependencies will transition to uninstalled and this equates to garbage collection.

5.2.14.1 State Precedence

When managing the life-cycle of shared resources, a state precedence is used. The order of states is as follows:

1. ACTIVE
2. RESOLVED
3. INSTALLED

4. UNINSTALLED

In the case of a subsystem resource, transitional states are not considered. Instead, the target end state of the transition is used.

Figure 14 shows an example of two features with a common bundle dependency C. Feature F1 is ACTIVE and so its content bundles A and C are active. Feature F2 is RESOLVED. It's content bundle B which is not shared is also RESOLVED, but because its content bundle C is shared with F1 and F1's state is higher in the order, bundle C is in the ACTIVE state.

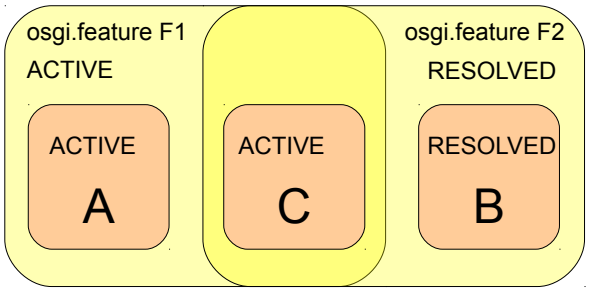

Figure 14: Shared resource state example

5.3 Subsystem Definitions

The following sections describe the headers used to describe subsystem definitions. As mentioned earlier, three types of subsystem are defined:

- Application – An implicitly scoped subsystem.
- Composite – An explicitly scoped subsystem.
- Feature – An unscoped subsystem.

~~A single set of headers is used by all types of subsystem with some headers only being valid for a subset of those types. A subsystem can exhibit some, or all, of the following features:~~

A single set of headers is used to describe all subsystem types with one header identifying the type. Some headers are only valid for particular subsystem types.

- ***Type*** – the type of the subsystem, such as application, composite or feature.
- ***Identity*** – a symbolic name and version (Note, this is does not define uniqueness, as the same subsystem may be installed multiple times, although not in the same scope (i.e. they cannot be peers)).

- Name – A human readable name for the subsystem.
- ***Description*** – human readable information about the subsystems
- ***Content*** – The things that make up the subsystem content, such as bundles, and configuration, and other subsystems.
- ***Bundle Sharing*** – ability to share bundles into or out of a subsystem.
- ***Package Sharing*** – the ability to share packages into or out of a subsystem. The defaults for packages sharing differ based on the type of subsystem (e.g. scoped vs unscoped).
- ***Service Sharing*** – the ability to share services into or out of a subsystem. The defaults for service sharing differ based on the type of subsystem (e.g. scoped vs unscoped).
- ***Nesting*** – the ability to define subsystems in terms of other subsystems.
- ***Transitive Closure*** – the ability to allow subsystems to define core content and have their remaining transitive dependencies calculated during deployment.
- ***Configuration*** – configuration for the subsystem and its contents.

A single set of subsystem headers is used to describe subsystems, with one header identifying the type of the subsystem.

5.3.1 Manifest Header Processing

Subsystem manifests follow the Jar manifest format, but to improve usability, there are two rules which are relaxed:

1. Line lengths can exceed the Jar manifest maximum of 72 bytes.
2. The last line is not required to be a newline.

5.3.2 Subsystem Manifest Headers

TODO: fit-n-finish on the headers (take a look at what we have for bundles and see what equivalents we need for subsystems.

5.3.2.1 Type Header

- `Subsystem-Type` – the type of the subsystem. This specification defines three subsystem types with the following values; `osgi.application`, `osgi.composite`, and `osgi.feature`. The default value is `osgi.application`. It is recommended that any new subsystem types have their names defined using a reverse domain naming scheme, e.g. `org.acme.subsystem.wibble`.

The following directive can be applied to the Subsystem-Type header:

- `provision-policy` – the provision-policy directive is used to describe additional qualities of the subsystem. This specification defines two values for the directive. 1) `root` – this means that the subsystem is considered to a be a root where its or its children's transitive resources can be provisioned. 2) `none` – there is no provision-policy and therefore the default subsystem behavior is unmodified. `None`

is the default for this directive.~~`acceptTransitive` – this means that the subsystem allows its transitive dependencies and those of its children to be provisioned inside it. 2) `default` – there is no provision-policy and therefore the default subsystem behavior is unmodified. `None` is the default for this directive.~~ The default value is `default`.

~~To allow for other policies, this directive is a comma separated list. It is logically a bit mask and therefore the default value of `default` need never be specified. Adding a policy value only modifies that aspect of the policy and therefore other aspects still have the default of `default`. Policies need not be specified in a particular order. Because comma is a reserved value, multivalued entries will need to be placed in quotes.~~

- Versioning Headers
- `Manifest-Version` – the version of the scheme for manifests that this manifest conforms to. This is the standard manifest header and is not defined by this specification. The default value is `1.0`.
- `Subsystem-ManifestVersion` – a~~n~~ subsystem manifest must conform to a version of the subsystem manifest headers. The version is defined by the OSGi specification and will be 1.0 for the first version.

5.3.2.2 Identity Headers

The following headers are used to specify the identity of asubsystem:

- `Subsystem-SymbolicName` – the symbolic name of the subsystem. Follows the same scheme and guidelines as `Bundle-SymbolicName`. To avoid confusion when viewing an environment that contains bundles and subsystems, it is recommended that different names be used for each. The default value is computed as described below.
- `Subsystem-Version` – the version of the subsystem (optional). Follows the same scheme and guidelines as `Bundle-Version`. The default value is `0.0.0` or computed as described below.

If not specified in the manifest, or the manifest is omitted, the following rules are defined for computing the default values of Subsystem-SymbolicName and Subsystem-Version. Manifest values override the defaults. If not otherwise specified, the default value of Subsystem-Version is always `0.0.0`. If the symbolic name is not provided in the manifest and cannot be computed by any of the following means, implementations may fail the installation.

1. When installing a subsystem via the subsystem interface (i.e. using one of the install methods), the following URI syntax must be used as the location in order to specify default values. Note that the optional URL component, if included, must be encoded.

   subsystem-uri ::= 'subsystem://' (url)? '?' symbolic-name ('#' version)?

   url ::= <see RFC 1738> // Retrieved via URI.getAuthority().

   symbolic-name ::= <see OSGi Core Specification 1.3.2> // Retrieved via URI.getQuery().

   version ::= <see OSGi Core Specification 3.2.5> // Retrieved via URI.getFragment().

2. When installing a subsystem with no content headers and containing other subsystem archives with no symbolic name or version, default values will be derived from the archive file names. The syntax is as follows.

subsystem-archive ::= symbolic-name ('@' version) '.ssa'

3. When installing a subsystem with content headers having clauses pointing to other subsystems with no symbolic name or version, the path and version attribute of the corresponding clause will be used as the default symbolic name and version.

5.3.2.3 Informational Headers

The following headers are informational and are intended to be human readable:

- `Subsystem-Name` – the human readable subsystem name. Follows the same scheme and guidelines as `Bundle-Name`. The default value is the symbolic name.

- `Subsystem-Description` – the description of the subsystem. Follows the same scheme and guidelines as `Bundle-Description`.

5.3.2.4 Content Header

The following header identifies the content of a subsystem:

- `Subsystem-Content` – the content of the subsystem. This is the key content for the subsystem. At runtime, this content will be provisioned as a set which may or may not be isolated, depending on the type of the subsystem. See section 5.3.2.6 for the detailed definition of subsystem content.

5.3.2.5 Dependency Header

The following header is used to ensure a particular Bundle or Subsystem is used to satisfy implicit package dependencies of a subsystem (e.g. packages used by application or feature subsystems):

- `Preferred-Provider` – a set of bundles or subsystems to be preferred to satisfy the subsystem's implicit package imports. See section 5.3.2.7 for the detailed definition of use bundle.

Preferred providers are used to express a preference for a bundle or subsystem to satisfy implicit package dependencies. If a~~n~~ subsystem has an implicit import package that is satisfied by a bundle listed in Preferred-Provider then the package import should be tied to that bundle (e.g. through bundle-symbolic-name or bundle-version matching attributes or equivalent mechanism). Any bundles listed in Preferred-Provider that satisfy implicit imports are provisioned into the subsystem root (as is the case with any other bundles satisfying implicit package dependencies).

This mechanism enables subsystems to share bundles or subsystems with other subsystems whilst still ensuring they use the provider of the packages they expect.

5.3.2.6 Subsystem-Content

Subsystem content is a list of content resources that can be of various types. The default type is bundle. A bundle is identified by its symbolic name. For example

```
Subsystem-Content: com.acme.bundles.bundle1,
 com.acme.subsystems.subsystem1;type:=osgi.subsystem
```

The default value of Subsystem-Content is computed from all resource files included in the subsystem archive. It is an error to omit the content header and have no resource files in the archive.

The following matching attribute can be applied to bundle content:

- `version` – a version range used to select the bundle version of the bundle to use. This follows the OSGi version range scheme, including the default value of 0.0.0. For composite subsystems, this value must be a fixed version range (e.g. "[1.0, 1.0]"). This is due to the fact that there is an inextricable link between the versions on the explicit import and export statements made on a composite and the chosen versions of the content bundles. Allowing variability in the content versions

The following directives can be applied to bundle content:

- `resolution` – states whether the bundle must exist in order for the subsystem to be considered complete. A value of `mandatory` (the default) means the bundle must exist, a value of `optional` means the subsystem can be complete even if the bundle does not exist.

- `type` – indicates the type of the content. This value is used to uniquely identify the content resource. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundles). The default type is 'osgi.bundle'. A subsystem resource is identified with a type of 'osgi.subsystem'.

5.3.2.7 Preferred-Provider

Preferred provider is a list of bundles and/or subsystems that the subsystem explicitly uses packages from. A bundle or subsystem is identified by its symbolic name and an optional type directive. For example

```
Preferred-Provider: com.acme.bundles.bundle1,
 com.acme.subsystems.subsystem1;type:=osgi.subsystem
```

The following matching attribute can be applied:

- `version` – a version range used to select the version of the bundle or subsystem to use. This follows the OSGi version range scheme, including the default value of 0.0.0.

The following directive can be applied to the Preferred-Provider header:

- `type` – indicates the type of the preferred provider. This value is used to uniquely identify the provider when resolving the subsystem. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundle). The default type is 'osgi.bundle'. A subsystem resource is identified with a type of 'osgi.subsystem'.

Open Question: Is there a need for service affinity or other types of capability?

5.3.2.8 Explicit Sharing Headers

A composite subsystem can make explicit statements about service, package and bundle sharing. In fact, in order to share anything, it must make these explicit statements. These statements are made using the following headers. Note, these headers are not valid for application or feature subsystems as any sharing they support is completely implicit.

- `Import-Package` – specifies a list of package constraints to import into the composite. Uses the same syntax as bundle Import-Package. Any exported package from a bundle or composite that is a peer to this composite and which matches one of the constraints is available to satisfy any Import-Package constraints of the content bundles.

- `Export-Package` – specifies a list of package to export out of the composite. Uses the same syntax as bundle Export-Package. Any exports listed here must exactly match (i.e. they must have the same attributes and directives, but they can appear in a different order) an export from one of the content bundles.

- `Require-Bundle` – specifies a list of require bundle constraints to import into the composite. Uses the same syntax as Require-Bundle. Any bundle that is a sibling to the composite which matches one of the constraints is available to satisfy Require-Bundle constraints of the composite's content bundles.

- `Subsystem-ImportService` – specifies a list of service interfaces and optional service filters that control the services that are imported into the composite subsystem. Any services registered by peer bundles or exported by peer composites that match one of these filters are made available to the content bundles. The syntax for this header is as follows:

```
Subsystem-ImportService ::= service-import ( ',' service-import )*
service-import ::= interface-name ( ';' filter )?
```

- `Subsystem-ExportService` – specifies a list of service filters that control the services that are exported out of the composite. Any services registered by content bundles that match one of these filters are made available to peer bundles and composites. The syntax for this header is as follows:

```
Subsystem-ExportService ::= service-export ( ',' service-export )*
service-export ::= interface-name ( ';' filter )?
```

5.3.3 Subsystem Archive

A subsystem archive is a zip file ending in the extension .ssa ~~file~~ (SubSystem Archive). Unlike jar archives, there are no special rules governing the order of the archive contents. A subsystem archive consists of the following:

1. The subsystem manifest. This is optional and if omitted will be calculated based on the rules defined in section Error: Reference source not found. The subsystem manifest is stored in the archive in a file named, OSGI-INF/SUBSYSTEM.MF.

2. A deployment manifest. This is optional and if omitted is calculated during provisioning of the subsystem, either by some management agent prior to calling **Subsystem.install** or as part of the **System.install** operation. It is anticipated that subsystem archives will rarely contain deployment manifests during development, but as the subsystem is tested prior to putting the subsystem into production, it will be common to add the deployment manifest to 'lock down' the bundles that are provisioned. The deployment manifest is stored in the archive in a file named, OSGI-INF/DEPLOYMENT.MF.

3. Any resources that may help with the provisioning of the subsystem. Note, it is feasible for a~~n~~ subsystem archive to contain no resources and have all it's contents and their dependencies provisioned from a repository. However, it is an error to install a subsystem whose archive contains no resources and no manifest.

5.3.4 ~~Defaulting Rules~~

~~TODO: Each header defaulted individually. Cover defaulting of things like:~~

- ~~symbolic name derived from archive name.~~

- ~~Version default to 0.0.0.0~~

- ~~Application name is archive name, including the extension.~~
- ~~etc~~

5.3.5 Manifest Localization

TODO: Describe how headers are localized. This will follow the approach used for bundle manifest headers.

5.3.6 Deployment Manifest

A deployment manifest describes the bundles and resources that should be provisioned for a particular subsystem definition. The deployment manifest can be packaged with the subsystem or determined during deployment. A deployment manifest could be authored by hand but this is highly unlikely as the process is complex and would be error prone. Therefore, in most cases the deployment manifest will be calculated by a resolver, either out of band in some pre-deployment or pre-installation step, or during the installation of the subsystem, the latter making use of the OBR specification, identified in the subsystem architecture in figure 4.

5.3.6.1 Portability

A deployment manifest describes the bundles and resources that need to be provisioned to satisfy the subsystem definition. Resolution happens in the context of a target runtime, which could be a specific server instance, an empty framework, and so on. As such, a deployment manifest may not be transferable from one environment to another.

Whilst the format and location of deployment manifests is standardized it is not a requirement that a deployment manifest calculated for one runtime work in another. Installation of a subsystem can therefore determine whether the deployment manifest is valid and if not, choose to calculate a new deployment, or fail the installation. If the supplied deployment manifest is not valid and a valid deployment manifest cannot be calculated, then the installation be failed. Failing an installation is done by throwing a SubsystemException from the install method.

5.3.6.2 Design

The deployment manifest describes the resources to be provisioned for a subsystem, irrespective of the subsystem type. These resources fall into two categories:

1. Deployed content: the content that is to be deployed into the subsystem.
2. Provision resource: the resources to be provisioned outside the subsystem in support of its external dependencies.

A deployment manifest also describes configuration for the subsystem package and service isolation in the form of headers for the packages and services that are imported or exported by the subsystem. In the case of feature subsystems, which have no isolation, these headers are not used as they would simply list all packages and services imported or exported by the feature subsystem contents.

A deployment manifest is located in the OSGI-INF folder of the Subsystem archive file and is named DEPLOYMENT.MF.

As with Subsystem manifests, Deployment manifests follow the Jar manifest format, but to improve usability, there are two rules which are relaxed:

1. Line lengths can exceed the Jar manifest maximum of 72 bytes.
2. The last line is not required to be a newline.

5.3.6.3 Manifest Version Header

- `Manifest-Version` – the version of the scheme for manifest that this manifest conforms to. The default value is 1.0.

5.3.6.4 Content Header

- `Deployed-Content` – the content to be deployed for the subsystem. This is the exact content that is provisioned into the Composite Bundle that implements the Subsystem. The default value is computed from the Subsystem-Content header and resolution results. Basically, the value is the same except version ranges are replaced with exact versions. The exact details of the Deployed-Content header are described in section 5.3.6.5.

5.3.6.5 Deployed-Content

Deployed content is a list of exact content that can be of various types. The default type is bundle, including composite bundle. A bundle is identified by its symbolic name. For example

```
Deployed-Content: com.acme.bundles.bundle1,
 com.acme.bundles.bundle2
```

Each entry must uniquely identify the resource to be provisioned into the subsystem.

The following matching attribute can be applied to bundle content:

- `deployed-version` – the exact version of the resource to be deployed. Deployed version is a specific version, not a version range, hence the use of a new attribute name.

The following directive can be applied to resource content:

- `type` – indicates the type of the content. The default value is “osgi.bundle”. This specification defines “osgi.subsystem” for subsystems.
- start-order – The precedence the resource should have during the start sequence. Resources with lower start-order values are started before resources with higher values. Resources with the same start-order value may be started sequentially or in parallel. The default value is “1”.

5.3.6.6 Subsystem Identification

The subsystem to which the deployment manifest applies is identified by the subsystem's symbolic name and version headers, defined earlier. The specific headers for each subsystem type are as follows:

- `Subsystem-SymbolicName` and `Subsystem-Version`

The default values match the values of the headers with the same names in the subsystem manifest. It is an error if the deployment manifest is contained in a subsystem archive and the identification does not match that of the archive (either in the subsystem's definition manifest or one which would be generated through the defaulting rules).

5.3.6.7 Provisioning Header

- `Provision-Resource` – the resources to be provisioned in support of the subsystem's transitive dependencies. These are the specific bundles that are provisioned outside the Composite Bundle that implements the Subsystem. The default value is computed from the set of all resources necessary to resolve the subsystem minus any content resources defined in the Subsystem-Content header. The exact details of the Provision-Resource header are described in section 5.3.6.8.

5.3.6.8 Provision-Resource

Provision resource lists the resources to be provisioned in support of a subsystem's external dependencies. These dependencies can include package or service requirements.

Provision resource has one required matching attribute:

- `deployed-version` – the exact version of the resource to be deployed. Deployed version is a specific version, not a version range, hence the use of a new attribute name.

Provision resource has the following optional directive:

- `type` – indicates the type of the resource. This value is used to uniquely identify the resource. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundles). The default type is 'osgi.bundle'. A subsystem resource is identified with a type of 'osgi.subsystem'.

5.3.6.9 Package Imports

Isolating subsystems (i.e. applications and composites) describe the exact packages they import in their deployment manifests. They do this using the bundle Import-Package header. Any packages that match the Import-Package statement must be allowed to pass into the subsystem as candidates for resolving the subsystem contents. Features subsystems provide no isolation or resolution affinity and therefore there is no need to described their package imports in the deployment manifest. For composite subsystems, these are the Import-Package statements taken directly for the subsystem manifest, but for application subsystems these must be calculated as described below.

5.3.6.10 Application Import-Package Calculation

Application resolution requires their resolution to prefer packages provided by content bundles over those provided outside the application. For this reason, the deployment manifest only lists Import-Package statements from the content bundles that are not satisfied when resolving the application contents in isolation.

5.3.6.11 Package Exports

Only composite subsystems explicitly export packages. Any Export-Package statements in the composite's SUBSYSTEM.MF are re-iterated in the deployment manifest. Feature subsystems essentially export all packages and therefore there is not need to describe every packages exported by the feature's bundles in this header.

5.3.6.12 Bundle Requirements

Isolating subsystems (i.e. applications and composites) describe their bundle requirements in their deployment manifests. They do this using the bundle Require-Bundle header. Any bundles that match the Require-Bundle statement are allowed to pass into the subsystem to resolve Require-Bundle requirements from the content bundles. Features subsystems provide no isolation or resolution affinity and therefore there is no need to described their bundle requirements in the deployment manifest. For composite subsystems, these are the Require-Bundle statements taken directly for the subsystem manifest, but for application subsystems these must be calculated as described below.

5.3.6.13 Application Require-Bundle Calculation

Application resolution requires their resolution to prefer content bundles over those provided outside the application. For this reason, the deployment manifest only lists Require-Bundle statements from the content bundles that are not satisfied when resolving the application contents in isolation.

5.3.6.14 Service Imports

Composite subsystems explicitly import services for use by their content bundles. Any entries described in the Subsystem-ImportService header are mapped to entires in the Deployed-ServiceImport header. The mapping takes the service-import entries of the syntax described in section 5.3.2.8, and combines them into a single filter. So

```
service-import ::= interface-name ( ';' filter )?
```

becomes

```
service-import ::= combined-filter

combined-filter ::= '(&(objectClass=' interface-name ')' filter ')'
```

For example

```
Subsystem-ImportService: com.acme.ServiceInterface;filter="(size=large)"
```

Becomes

```
Deployed-ServiceImport: (&(objectClass=com.acme.ServiceInterface)(size=large))
```

Application subsystems do not explicitly import services. The services required to be imported into an application are essentially those required by the application's content bundles that are not also provided by the content bundles. There is no standard way to determine this, but a subsystem runtime is permitted to use it's own means to determine the services to import. Examples include resource metadata from a bundle repository, or analysis of bundle contents (e.g. Blueprint or Declarative Service XMLs). A subsystem runtime is then free to add Deployed-ServiceImport entries in order to complete the application's deployment manifest definition.

Feature subsystems do not isolate services and therefore there is no need to declare the services that are imported into the feature. All services outside the feature are available for use inside the feature.

5.3.6.15 Service Exports

Composite subsystems explicitly export services for use by peer bundles or subsystems. Any entries described in the Subsystem-ExportService header are mapped to entires in the Deployed-ServiceExport header. The mapping takes the service-export entries of the syntax described in section 5.3.2.8, and combines them into a single filter. So

```
service-export ::= interface-name ( ';' filter )?
```

becomes

```
service-export ::= combined-filter
combined-filter ::= '(&(objectClass=' interface-name ')' filter ')'
```

For example

```
Subsystem-ExportService: com.acme.ServiceInterface;filter="(size=large)"
```

Becomes

```
Deployed-ServiceExport: (&(objectClass=com.acme.ServiceInterface)(size=large))
```

Application subsystems do not export services outside the application and therefore it is an error to include the Deployed-ServiceExport header in an application deployment manifest.

Feature subsystems do not isolate services and therefore there is no need to declare the services that are exported into the feature. All services inside the feature are available for use outside the feature.

5.3.6.16 Provision-Resource

```
Manifest-Version: 1.0
SubsystemApplication-SymbolicName: com.ibm.ws.eba.example.blog.app
SubsystemApplication-Version: 1.0.0
Deployed-Content: com.ibm.ws.eba.example.blog.api;deployed-version=1.0
 .0,com.ibm.ws.eba.example.blog.persistence;deployed-version=1.0.0,com
 .ibm.ws.eba.example.blog.web;deployed-version=1.0.0,com.ibm.ws.eba.ex
 ample.blog.biz;deployed-version=1.0.0
Deployed-ServiceImport: (objectClass=com.ibm.ws.eba.example.blog.com
 ment.persistence.api.BlogCommentService)
Import-Package: com.ibm.json.java;version="1.0.0";bundle-symbolic-name
 ="com.ibm.json.java";bundle-version="[1.0.0,1.0.0]",javax.persistence
 ;version="1.0.0",javax.servlet.http;version="[2.5.0,3.0.0)",javax.ser
 vlet;version="[2.5.0,3.0.0)"
Provision-Resource: com.ibm.json.java;deployed-version=1.0.0
```

Figure 15: Example Deployment Manifest.

5.3.6.17 Deployment Manifest Creation

5.3.6.18 ~~Deployment Manifest Creation~~

5.3.6.19

5.3.7 Configuration

Notes:

- Config Admin
- Security Permissions

5.4 Transitive Closure

Notes:

- Desire expressed to be able to control 'auto-provisioning'. Some debate as to whether that is expressed in the subsystem artefact or a policy of the environment into which the subsystem is deployed. The latter seemed more appropriate. - Management Agent can choose to do this or not.
- Need to be able to handle runtime requirements
- Need to be able to handle license requirements

5.4.1

5.5

6 JavaDoc

OSGi Javadoc

8/29/11 3:53 PM

Package Summary		Page
org.osgi.service.subsystem	Subsystem Package Version 1.0.	46

Package org.osgi.service.subsystem

Subsystem Package Version 1.0.

See:

Description

Interface Summary		Page
Subsystem	A representation of a subsystem in the framework.	*47*

Class Summary		Page
SubsystemConstants	Defines the constants used by subsystems.	*57*

Enum Summary		Page
Subsystem.State	The states of a subsystem in the framework.	*54*
SubsystemConstants.EVENT_TYPE	The subsystem lifecycle event types that can be produced by a subsystem.	*64*

Exception Summary		Page
SubsystemException	Exception thrown by Subsystem when a problem occurs.	*68*

Package org.osgi.service.subsystem Description

Subsystem Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.subsystem; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.subsystem; version="[1.0,1.1)"
```

Interface Subsystem

org.osgi.service.subsystem

public interface **Subsystem**

A representation of a subsystem in the framework. A subsystem is a collection of bundles and/or other resource. A subsystem has isolation semantics. Subsystem types are defined that have different default isolation semantics. For example, an Application subsystem does not export any of the packages or services provided by its content bundles, and imports any packages or services that are required to satisfy unresolved package or service dependencies of the content bundles. A subsystem is defined using a manifest format.

ThreadSafe

Nested Class Summary		Page
static enum	**Subsystem.State** The states of a subsystem in the framework.	54

Method Summary		Page
void	**cancel**() Cancels the currently executing asynchronous life-cycle operation, if any.	48
Collection<Subsystem>	**getChildren**() Gets the subsystems managed by this service.	48
Collection<org.osgi.framework.resource.Resource>	**getConstituents**() Returns a snapshot of all Resources currently constituting this Subsystem.	48
Map<String,String>	**getHeaders**() Gets the headers used to define this subsystem.	48
Map<String,String>	**getHeaders**(String locale) Gets the headers used to define this subsystem.	49
String	**getLocation**() The location identifier used to install this subsystem through Subsystem.install.	49
Subsystem	**getParent**() Gets the parent Subsystem that scopes this subsystem instance.	49
Subsystem.State	**getState**() Gets the state of the subsystem.	49
long	**getSubsystemId**() Gets the identifier of the subsystem.	50
String	**getSymbolicName**() Gets the symbolic name of this subsystem.	50
org.osgi.framework.Version	**getVersion**() Gets the version of this subsystem.	50
Subsystem	**install**(String location) Install a new subsystem from the specified location identifier.	50
Subsystem	**install**(String location, InputStream content) Install a new subsystem from the specified InputStream object.	51
void	**start**() Starts the subsystem.	52

void	**stop**() Stops the subsystem.	52
void	**uninstall**() Uninstall the given subsystem.	52

Method Detail

cancel

```
void cancel()
     throws SubsystemException
```

Cancels the currently executing asynchronous life-cycle operation, if any.

Throws:

SubsystemException - - If this subsystem is not in one of the transitional states or the currently executing operation cannot be cancelled for any reason.

---

getChildren

```
Collection<Subsystem> getChildren()
```

Gets the subsystems managed by this service. This only includes the top-level Subsystems installed in the Framework, CoompositeBundle or Subsystem from which this service has been retrieved.

Returns:

The Subsystems managed by this service.

Throws:

IllegalStateException - If the subsystem is in the installing state or transitioned to the uninstalled state due to a failed installation.

---

getConstituents

```
Collection<org.osgi.framework.resource.Resource> getConstituents()
```

Returns a snapshot of all Resources currently constituting this Subsystem. If this Subsystem has no Resources, the Collection will be empty.

Returns:

A snapshot of all Resources currently constituting this Subsystem.

Throws:

IllegalStateException - If the subsystem is in the installing state or transitioned to the uninstalled state due to a failed installation.

---

getHeaders

```
Map<String,String> getHeaders()
```

Gets the headers used to define this subsystem. The headers will be localized using the locale returned by java.util.Locale.getDefault. This is equivalent to calling getHeaders(null).

Returns:
The headers used to define this subsystem.
Throws:
`SecurityException` - If the caller does not have the appropriate AdminPermission[this,METADATA] and the runtime supports permissions.
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

---

getHeaders

`Map<String,String> **getHeaders**(String locale)`

Gets the headers used to define this subsystem.

Parameters:
`locale` - The locale name to be used to localize the headers. If the locale is null then the locale returned by java.util.Locale.getDefault is used. If the value is the empty string then the returned headers are returned unlocalized.
Returns:
the headers used to define this subsystem, localized to the specified locale.
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

---

getLocation

`String **getLocation**()`

The location identifier used to install this subsystem through Subsystem.install. This identifier does not change while this subsystem remains installed, even after Subsystem.update. This location identifier is used in Subsystem.update if no other update source is specified.

Returns:
The string representation of the subsystem's location identifier.

---

getParent

`Subsystem **getParent**()`

Gets the parent Subsystem that scopes this subsystem instance.

Returns:
The Subsystem that scopes this subsystem or null if there is no parent subsystem (e.g. if the outer scope is the framework).
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

---

getState

`Subsystem.State **getState**()`

Gets the state of the subsystem.

Returns:
The state of the subsystem.

---

getSubsystemId

`long` **`getSubsystemId`**`()`

Gets the identifier of the subsystem. Subsystem identifiers are assigned when the subsystem is installed and are unique within the framework.

Returns:
The identifier of the subsystem.

---

getSymbolicName

`String` **`getSymbolicName`**`()`

Gets the symbolic name of this subsystem.

Returns:
The symbolic name of this subsystem.
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

---

getVersion

`org.osgi.framework.Version` **`getVersion`**`()`

Gets the version of this subsystem.

Returns:
The version of this subsystem.
Throws:
`IllegalStateException` - If the subsystem is in the `installing state` or transitioned to the `uninstalled state` due to a failed installation.

---

install

`Subsystem` **`install`**`(String location)`
`throws SubsystemException`

Install a new subsystem from the specified location identifier.

This method performs the same function as calling install(String, InputStream) with the specified location identifier and a null InputStream.

Parameters:
`location` - The location identifier of the subsystem to be installed.
Returns:
The installed subsystem.
Throws:
`SubsystemException` - If the subsystem could not be installed for any reason.

SecurityException - If the caller does not have the appropriate AdminPermission[installed subsystem,LIFECYCLE], and the Java Runtime Environment supports permissions.

---

install

```
Subsystem install(String location,
                  InputStream content)
           throws SubsystemException
```

Install a new subsystem from the specified InputStream object.

If the specified InputStream is null, the InputStream must be created from the specified location.

The specified location identifier will be used as the identity of the subsystem. Every installed subsystem is uniquely identified by its location identifier which is typically in the form of a URL.

TODO: Understand whether this all change when we can install the same bundle multiple times.

A subsystem and its contents must remain installed across Framework and VM restarts. The subsystem itself is installed atomically, however its contents are not.

The following steps are required to install a subsystem:

10. If there is an existing subsystem containing the same location identifier as the subsystem to be installed, then the existing subsystem is returned.
11. If this is a new install, then a new Subsystem is created with its id set to the next available value (ascending order).
12. The subsystem's state is set to INSTALLING and if EventAdmin is available, an event of type INSTALLING is fired.
13. The following installation steps are then started and performed asynchronously and the new subsystem is returned to the caller.
14. The subsystem content is read from the input stream.
15. If the subsystem requires isolation (i.e. is an application or a composite), then isolation is set up while the install is in progress, such that none of the content bundles can be resolved. This isolation is not changed until the subsystem is explicitly requested to resolve (i.e. as a result of a Subsystem.start() operation).
16. If the subsystem does not include a deployment manifest, then the subsystem runtime must calculate one.
17. The resources identified in the deployment manifest are installed into the framework. All content resources are installed into the Subsystem, whereas transitive dependencies are installed into an ancestor subsystem. If any resources fail to install, then the entire installation is failed. Transitive resources are free to resolve and start independent of the subsystem they were installed for.
18. The subsystem's state is set to INSTALLED and if EventAdmin is available an INSTALLED event is fired.

Parameters:

location - The location identifier of the subsystem to be installed.

content - The InputStream from where the subsystem is to be installed or null if the location is to be used to create the InputStream.

Returns:

The installed subsystem.

Throws:

SubsystemException - If the subsystem could not be installed for any reason.

SecurityException - If the caller does not have the appropriate AdminPermission[installed subsystem,LIFECYCLE], and the Java Runtime Environment supports permissions.

---

start

```
void start()
    throws SubsystemException
```

Starts the subsystem. The subsystem is started according to the rules defined for Bundles and the content bundles are enabled for activation.

Throws:

SubsystemException - If this subsystem could not be started.
IllegalStateException - If this subsystem has been uninstalled.
SecurityException - If the caller does not have the appropriate AdminPermission[this,EXECUTE] and the runtime supports permissions.

---

stop

```
void stop()
         throws SubsystemException
```

Stops the subsystem. The subsystem is stopped according to the rules defined for Bundles and the content bundles are disabled for activation and stopped.

Throws:

SubsystemException - If an internal exception is thrown while stopping the subsystem (e.g. a BundleException from Bundle.stop).
IllegalStateException - - If this subsystem has been uninstalled.
SecurityException - - If the caller does not have the appropriate AdminPermission[this,EXECUTE] and the runtime supports permissions.

---

uninstall

```
void uninstall()
        throws SubsystemException
```

Uninstall the given subsystem.

This method causes the Framework to notify other bundles and subsystems that this subsystem is being uninstalled, and then puts this subsystem into the UNINSTALLED state. The Framework must remove any resources related to this subsystem that it is able to remove. If this subsystem has exported any packages, the Framework must continue to make these packages available to their importing bundles or subsystems until the org.osgi.service.packageadmin.PackageAdmin.refreshPackages(org.osgi.framework.Bundle[]) method has been called or the Framework is relaunched. The following steps are required to uninstall a subsystem:

1. If this subsystem's state is UNINSTALLED then an IllegalStateException is thrown.
2. If this subsystem's state is ACTIVE, STARTING or STOPPING, this subsystem is stopped as described in the Subsystem.stop() method. If Subsystem.stop() throws an exception, a Framework event of type FrameworkEvent.ERROR is fired containing the exception.
3. This subsystem's state is set to UNINSTALLED.
4. A subsystem event of type SubsystemEvent.UNINSTALLED is fired.
5. This subsystem and any persistent storage area provided for this subsystem by the Framework are removed.

Throws:

SubsystemException - If the uninstall failed.
IllegalStateException - If the subsystem is already in the UNISTALLED state.

`SecurityException` - If the caller does not have the appropriate AdminPermission[this,LIFECYCLE] and the Java Runtime Environment supports permissions.

Enum Subsystem.State

org.osgi.service.subsystem

```
java.lang.Object
  └ java.lang.Enum<Subsystem.State>
      └ org.osgi.service.subsystem.Subsystem.State
```

All Implemented Interfaces:

Comparable<Subsystem.State>, Serializable

Enclosing class:

Subsystem

```
public static enum Subsystem.State
extends Enum<Subsystem.State>
```

The states of a subsystem in the framework. These states match those of a Bundle and are derived using the same rules as CompositeBundles. As such, they are more a reflection of what content bundles are permitted to do rather than an aggregation of the content bundle states.

Enum Constant Summary	Page
ACTIVE A subsystem is in the ACTIVE state when it has reached the beginning start-level (for starting it's contents), and all its persistently started content bundles that are resolved and have had their start-levels met have completed, or failed, their activator start method.	55
INSTALLED A subsystem is in the INSTALLED state when all resources are successfully installed.	55
INSTALLING A subsystem is in the INSTALLING state when it is initially created.	55
RESOLVED Â A subsystem is in the RESOLVED state when all resources are resolved.	55
RESOLVING Â A subsystem in the RESOLVING is allowed to have its content bundles resolved.	55
STARTING A subsystem is in the STARTING state when all its content bundles are enabled for activation.	55
STOPPING Â A subsystem in the STOPPING state is in the process of taking its its active start level to zero, stopping all the content bundles.	55
UNINSTALLED A subsystem is in the UNINSTALLED state when all its content bundles and uninstalled and its system bundle context is invalidated.	56
UNINSTALLING	56
UPDATING	56

Method Summary		Page
static Subsystem.State	**valueOf**(String name)	56
static Subsystem.State[]	**values**()	56

Enum Constant Detail

INSTALLING

```
public static final Subsystem.State INSTALLING
```

A subsystem is in the INSTALLING state when it is initially created.

INSTALLED

```
public static final Subsystem.State INSTALLED
```

A subsystem is in the INSTALLED state when all resources are successfully installed.

RESOLVING

```
public static final Subsystem.State RESOLVING
```

Â A subsystem in the RESOLVING is allowed to have its content bundles resolved.

RESOLVED

```
public static final Subsystem.State RESOLVED
```

Â A subsystem is in the RESOLVED state when all resources are resolved.

STARTING

```
public static final Subsystem.State STARTING
```

A subsystem is in the STARTING state when all its content bundles are enabled for activation.

ACTIVE

```
public static final Subsystem.State ACTIVE
```

A subsystem is in the ACTIVE state when it has reached the beginning start-level (for starting it's contents), and all its persistently started content bundles that are resolved and have had their start-levels met have completed, or failed, their activator start method.

STOPPING

```
public static final Subsystem.State STOPPING
```

Â A subsystem in the STOPPING state is in the process of taking its its active start level to zero, stopping all the content bundles.

UPDATING

```
public static final Subsystem.State UPDATING
```

UNINSTALLING

```
public static final Subsystem.State UNINSTALLING
```

UNINSTALLED

```
public static final Subsystem.State UNINSTALLED
```

A subsystem is in the UNINSTALLED state when all its content bundles and uninstalled and its system bundle context is invalidated.

Method Detail

values

```
public static Subsystem.State[] values()
```

---

valueOf

```
public static Subsystem.State valueOf(String name)
```

Class SubsystemConstants

org.osgi.service.subsystem

```
java.lang.Object
  └ org.osgi.service.subsystem.SubsystemConstants
```

```
public class SubsystemConstants
extends Object
```

Defines the constants used by subsystems.

Nested Class Summary		Page
static enum	SubsystemConstants.EVENT_TYPE The subsystem lifecycle event types that can be produced by a subsystem.	64

Field Summary		Page
static String	DEPLOYED_CONTENT Manifest header identifying the resources to be deployed.	58
static String	DEPLOYED_VERSION_ATTRIBUTE Manifest header attribute identifying the deployed version.	59
static String	EVENT_SUBSYSTEM_ID Key for the event property that holds the subsystem id.	59
static String	EVENT_SUBSYSTEM_LOCATION Key for the event property that holds the subsystem location.	59
static String	EVENT_SUBSYSTEM_SYMBOLICNAME Key for the event property that holds the subsystem symbolic name.	59
static String	EVENT_SUBSYSTEM_VERSION Key for the event property that holds the subsystem version.	59
static String	EVENT_SUBYSTEM_STATE Key for the event property that holds the subsystem state.	59
static String	EVENT_TOPIC The topic for subsystem event admin events.	59
static String	EVENT_TOPIC_INTERNALS The topic for subsystem internal event admin events.	59
static String	EXPORT_PACKAGE Manifest header identifying packages offered for export.	60
static String	IDENTITY_TYPE_ATTRIBUTE Manifest header attribute identifying the resource type.	60
static String	IDENTITY_TYPE_BUNDLE Manifest header attribute value identifying a bundle resource type.	60
static String	IDENTITY_TYPE_SUBSYSTEM Manifest header attribute value identifying a subsystem resource type.	60
static String	IMPORT_PACKAGE Manifest header identifying packages required for import.	60
static String	PREFERRED_PROVIDER Manifest header used to express a preference for particular resources to satisfy implicit package dependencies.	60

static String	**PROVISION_RESOURCE** Manifest header identifying the resources to be deployed to satisfy the transitive dependencies of a subsystem.	61
static String	**REQUIRE_BUNDLE** Manifest header identifying symbolic names of required bundles.	61
static String	**RESOLUTION_DIRECTIVE** Manifest header directive identifying the resolution type.	61
static String	**RESOLUTION_MANDATORY** Manifest header directive value identifying a mandatory resolution type.	61
static String	**RESOLUTION_OPTIONAL** Manifest header directive value identifying an optional resolution type.	61
static String	**START_LEVEL_DIRECTIVE** Manifest header directive identifying the start level.	61
static String	**SUBSYSTEM_CONTENT** The list of subsystem contents identified by a symbolic name and version.	61
static String	**SUBSYSTEM_DESCRIPTION** Human readable description.	62
static String	**SUBSYSTEM_EXPORTSERVICE** Manifest header identifying services offered for export.	62
static String	**SUBSYSTEM_IMPORTSERVICE** Manifest header identifying services required for import.	62
static String	**SUBSYSTEM_MANIFESTVERSION** The subsystem manifest version header must be present and equals to 1.0 for this version of applications.	62
static String	**SUBSYSTEM_NAME** Human readable application name.	62
static String	**SUBSYSTEM_SYMBOLICNAME** Symbolic name for the application.	62
static String	**SUBSYSTEM_TYPE** Manifest header identifying the subsystem type.	62
static String	**SUBSYSTEM_TYPE_APPLICATION** Manifest header value identifying an application subsystem.	63
static String	**SUBSYSTEM_TYPE_COMPOSITE** Manifest header value identifying a composite subsystem.	63
static String	**SUBSYSTEM_TYPE_FEATURE** Manifest header value identifying a feature subsystem.	63
static String	**SUBSYSTEM_VERSION** Version of the application.	63
static String	**VERSION_ATTRIBUTE** Manifest header attribute indicating a version or version range.	63

Field Detail

DEPLOYED_CONTENT

```
public static final String DEPLOYED_CONTENT = "Deployed-Content"
```

Manifest header identifying the resources to be deployed.

DEPLOYED_VERSION_ATTRIBUTE

```
public static final String DEPLOYED_VERSION_ATTRIBUTE = "deployed-version"
```

Manifest header attribute identifying the deployed version.

EVENT_SUBSYSTEM_ID

```
public static final String EVENT_SUBSYSTEM_ID = "subsystem.id"
```

Key for the event property that holds the subsystem id.

EVENT_SUBSYSTEM_LOCATION

```
public static final String EVENT_SUBSYSTEM_LOCATION = "subsystem.location"
```

Key for the event property that holds the subsystem location.

EVENT_SUBYSTEM_STATE

```
public static final String EVENT_SUBYSTEM_STATE = "subsystem.state"
```

Key for the event property that holds the subsystem state.

EVENT_SUBSYSTEM_SYMBOLICNAME

```
public static final String EVENT_SUBSYSTEM_SYMBOLICNAME = "subsystem.symbolicname"
```

Key for the event property that holds the subsystem symbolic name.

EVENT_SUBSYSTEM_VERSION

```
public static final String EVENT_SUBSYSTEM_VERSION = "subsystem.version"
```

Key for the event property that holds the subsystem version.

EVENT_TOPIC

```
public static final String EVENT_TOPIC = "org/osgi/service/Subsystem/"
```

The topic for subsystem event admin events.

EVENT_TOPIC_INTERNALS

```
public static final String EVENT_TOPIC_INTERNALS = "org/osgi/service/SubsystemInternals/"
```

The topic for subsystem internal event admin events.

EXPORT_PACKAGE

```
public static final String EXPORT_PACKAGE = "Export-Package"
```

Manifest header identifying packages offered for export.

See Also:
org.osgi.framework.Constants.EXPORT_PACKAGE

IDENTITY_TYPE_ATTRIBUTE

```
public static final String IDENTITY_TYPE_ATTRIBUTE = "type"
```

Manifest header attribute identifying the resource type. The default value is IDENTITY_TYPE_BUNDLE.

See Also:
org.osgi.framework.resource.ResourceConstants.IDENTITY_TYPE_ATTRIBUTE

IDENTITY_TYPE_BUNDLE

```
public static final String IDENTITY_TYPE_BUNDLE = "osgi.bundle"
```

Manifest header attribute value identifying a bundle resource type.

See Also:
org.osgi.framework.resource.ResourceConstants.IDENTITY_TYPE_BUNDLE

IDENTITY_TYPE_SUBSYSTEM

```
public static final String IDENTITY_TYPE_SUBSYSTEM = "osgi.subsystem"
```

Manifest header attribute value identifying a subsystem resource type.

IMPORT_PACKAGE

```
public static final String IMPORT_PACKAGE = "Import-Package"
```

Manifest header identifying packages required for import.

See Also:
org.osgi.framework.Constants.IMPORT_PACKAGE

PREFERRED_PROVIDER

```
public static final String PREFERRED_PROVIDER = "Preferred-Provider"
```

Manifest header used to express a preference for particular resources to satisfy implicit package dependencies.

PROVISION_RESOURCE

```
public static final String PROVISION_RESOURCE = "Provision-Resource"
```

Manifest header identifying the resources to be deployed to satisfy the transitive dependencies of a subsystem.

REQUIRE_BUNDLE

```
public static final String REQUIRE_BUNDLE = "Require-Bundle"
```

Manifest header identifying symbolic names of required bundles.

RESOLUTION_DIRECTIVE

```
public static final String RESOLUTION_DIRECTIVE = "resolution"
```

Manifest header directive identifying the resolution type. The default value is RESOLUTION_MANDATORY.

See Also:
org.osgi.framework.Constants.RESOLUTION_DIRECTIVE

RESOLUTION_MANDATORY

```
public static final String RESOLUTION_MANDATORY = "mandatory"
```

Manifest header directive value identifying a mandatory resolution type.

See Also:
org.osgi.framework.Constants.RESOLUTION_MANDATORY

RESOLUTION_OPTIONAL

```
public static final String RESOLUTION_OPTIONAL = "optional"
```

Manifest header directive value identifying an optional resolution type.

See Also:
org.osgi.framework.Constants.RESOLUTION_OPTIONAL

START_LEVEL_DIRECTIVE

```
public static final String START_LEVEL_DIRECTIVE = "start-level"
```

Manifest header directive identifying the start level.

SUBSYSTEM_CONTENT

```
public static final String SUBSYSTEM_CONTENT = "Subsystem-Content"
```

The list of subsystem contents identified by a symbolic name and version.

SUBSYSTEM_DESCRIPTION

```
public static final String SUBSYSTEM_DESCRIPTION = "Subsystem-Description"
```

Human readable description.

SUBSYSTEM_EXPORTSERVICE

```
public static final String SUBSYSTEM_EXPORTSERVICE = "Subsystem-ExportService"
```

Manifest header identifying services offered for export.

SUBSYSTEM_IMPORTSERVICE

```
public static final String SUBSYSTEM_IMPORTSERVICE = "Subsystem-ImportService"
```

Manifest header identifying services required for import.

SUBSYSTEM_MANIFESTVERSION

```
public static final String SUBSYSTEM_MANIFESTVERSION = "Subsystem-ManifestVersion"
```

The subsystem manifest version header must be present and equals to 1.0 for this version of applications.

SUBSYSTEM_NAME

```
public static final String SUBSYSTEM_NAME = "Subsystem-Name"
```

Human readable application name.

SUBSYSTEM_SYMBOLICNAME

```
public static final String SUBSYSTEM_SYMBOLICNAME = "Subsystem-SymbolicName"
```

Symbolic name for the application. Must be present.

SUBSYSTEM_TYPE

```
public static final String SUBSYSTEM_TYPE = "Subsystem-Type"
```

Manifest header identifying the subsystem type.

SUBSYSTEM_TYPE_APPLICATION

```
public static final String SUBSYSTEM_TYPE_APPLICATION = "osgi.application"
```

Manifest header value identifying an application subsystem.

SUBSYSTEM_TYPE_COMPOSITE

```
public static final String SUBSYSTEM_TYPE_COMPOSITE = "osgi.composite"
```

Manifest header value identifying a composite subsystem.

SUBSYSTEM_TYPE_FEATURE

```
public static final String SUBSYSTEM_TYPE_FEATURE = "osgi.feature"
```

Manifest header value identifying a feature subsystem.

SUBSYSTEM_VERSION

```
public static final String SUBSYSTEM_VERSION = "Subsystem-Version"
```

Version of the application. If not present, the default value is 0.0.0.

VERSION_ATTRIBUTE

```
public static final String VERSION_ATTRIBUTE = "version"
```

Manifest header attribute indicating a version or version range. The default value is `org.osgi.framework.Version.emptyVersion`.

Enum SubsystemConstants.EVENT_TYPE

org.osgi.service.subsystem

```
java.lang.Object
  └ java.lang.Enum<SubsystemConstants.EVENT_TYPE>
      └ org.osgi.service.subsystem.SubsystemConstants.EVENT_TYPE
```

All Implemented Interfaces:

Comparable<SubsystemConstants.EVENT_TYPE>, Serializable

Enclosing class:

SubsystemConstants

```
public static enum SubsystemConstants.EVENT_TYPE
extends Enum<SubsystemConstants.EVENT_TYPE>
```

The subsystem lifecycle event types that can be produced by a subsystem. See ? and Subsystem for details on the circumstances under which these events are fired.

Enum Constant Summary	Page
CANCELED Event type used to indicate that the operations was cancelled (e.g.	66
CANCELING Event type used to indicate that a subsystem operation is being cancelled.	66
FAILED Event type used to indicate that the operation failed (e.g.	67
INSTALLED Event type used to indicate a subsystem has been installed.	65
INSTALLING Event type used to indicate a subsystem is installing.	65
RESOLVED Event type used to indicate a subsystem has been resolved.	65
RESOLVING Event type used to indicate a subsystem is resolving.	65
STARTED Event type used to indicate a subsystem has been started.	65
STARTING Event type used to indicate a subsystem is starting.	65
STOPPED Event type used to indicate a subsystem has been stopped.	66
STOPPING Event type used to indicate a subsystem is stopping.	66
UNINSTALLED Event type used to indicate a subsystem has been uninstalled.	66
UNINSTALLING Event type used to indicate a subsystem is uninstalling.	66
UPDATED Event type used to indicate a subsystem has been updated.	66
UPDATING Event type used to indicate a subsystem is updating.	66

Method Summary		Page
static SubsystemConstants.EVENT_TYPE	**valueOf**(String name)	67
static SubsystemConstants.EVENT_TYPE[]	**values**()	67

Enum Constant Detail

INSTALLING

public static final SubsystemConstants.EVENT_TYPE **INSTALLING**

Event type used to indicate a subsystem is installing.

INSTALLED

public static final SubsystemConstants.EVENT_TYPE **INSTALLED**

Event type used to indicate a subsystem has been installed.

RESOLVING

public static final SubsystemConstants.EVENT_TYPE **RESOLVING**

Event type used to indicate a subsystem is resolving.

RESOLVED

public static final SubsystemConstants.EVENT_TYPE **RESOLVED**

Event type used to indicate a subsystem has been resolved.

STARTING

public static final SubsystemConstants.EVENT_TYPE **STARTING**

Event type used to indicate a subsystem is starting.

STARTED

public static final SubsystemConstants.EVENT_TYPE **STARTED**

Event type used to indicate a subsystem has been started.

STOPPING

public static final SubsystemConstants.EVENT_TYPE **STOPPING**

Event type used to indicate a subsystem is stopping.

STOPPED

public static final SubsystemConstants.EVENT_TYPE **STOPPED**

Event type used to indicate a subsystem has been stopped.

UPDATING

public static final SubsystemConstants.EVENT_TYPE **UPDATING**

Event type used to indicate a subsystem is updating.

UPDATED

public static final SubsystemConstants.EVENT_TYPE **UPDATED**

Event type used to indicate a subsystem has been updated.

UNINSTALLING

public static final SubsystemConstants.EVENT_TYPE **UNINSTALLING**

Event type used to indicate a subsystem is uninstalling.

UNINSTALLED

public static final SubsystemConstants.EVENT_TYPE **UNINSTALLED**

Event type used to indicate a subsystem has been uninstalled.

CANCELING

public static final SubsystemConstants.EVENT_TYPE **CANCELING**

Event type used to indicate that a subsystem operation is being cancelled.

CANCELED

public static final SubsystemConstants.EVENT_TYPE **CANCELED**

Event type used to indicate that the operations was cancelled (e.g. an install was cancelled).

FAILED

```
public static final SubsystemConstants.EVENT_TYPE FAILED
```

Event type used to indicate that the operation failed (e.g. an exception was thrown during installation).

Method Detail

values

```
public static SubsystemConstants.EVENT_TYPE[] values()
```

valueOf

```
public static SubsystemConstants.EVENT_TYPE valueOf(String name)
```

Class SubsystemException

org.osgi.service.subsystem

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ org.osgi.service.subsystem.SubsystemException
```

All Implemented Interfaces:
Serializable

```
public class SubsystemException
extends RuntimeException
```

Exception thrown by Subsystem when a problem occurs.

Constructor Summary	Page
SubsystemException() Construct a subsystem exception with no message.	68
SubsystemException(String message) Construct a subsystem exception specifying a message.	68
SubsystemException(String message, Throwable cause) Construct a subsystem exception specifying a message and wrapping an existing exception.	69
SubsystemException(Throwable cause) Construct a subsystem exception wrapping an existing exception.	68

Constructor Detail

SubsystemException

```
public SubsystemException()
```

Construct a subsystem exception with no message.

SubsystemException

```
public SubsystemException(String message)
```

Construct a subsystem exception specifying a message.

Parameters:
`message` - The message to include in the exception.

SubsystemException

```
public SubsystemException(Throwable cause)
```

Construct a subsystem exception wrapping an existing exception.

Parameters:

`cause` - The cause of the exception.

SubsystemException

```
public SubsystemException(String message,
                          Throwable cause)
```

Construct a subsystem exception specifying a message and wrapping an existing exception.

Parameters:

`message` - The message to include in the exception.
`cause` - The cause of the exception.

---

7 State Tables

7.1 State lock with cancelable operations

The following approach treats operations as uninterruptable with the exception of the explicit use of cancel:

1. Transitional states lock out other operations - causes them to be queued up.
2. If a queued operation cannot acquire the lock within a 'reasonable time period' then an exception is thrown for that operation.
3. Transitional states can be canceled through the subsystem cancel() operation which ~~uses the coordinator and resource processors to~~ tries~~y~~ to return the subsystem to its previous state.
4. Errors are typically IllegalStateExceptions.
5. No-ops are shown by blank cells.

	cancel	install	uninstall	start	stop	update
~~CREATED?~~		~~Install~~	~~Error~~	~~Install, Resolve and Start~~	~~Error~~	~~Error~~
INSTALLING	~~Fail the Coordination (end up in CREATED? - bundles in UNINSTALLED?)~~ If installing aborted, UNINSTALLED. Otherwise, error.		Wait for Install to complete, then start Uninstall.	Wait for Install to complete then Start.		Wait for Install to complete then Update.
INSTALLED	Error		Uninstall	Resolve and Start.		Update
RESOLVING	If resolving aborted, INSTALLED. Otherwise, error.		Uninstall	Wait for resolve to complete then Start.		Wait for Resolve to complete then Update.
RESOLVED	Error		Uninstall	Start		Update
STARTING	~~Fail the Coordination (end up in RESOLVED)~~If starting aborted, RESOLVED. Otherwise, error.		Wait for Start to complete then uninstall.		Wait for Start to complete then Stop.	Wait for Start to complete then Update.
ACTIVE	Error		Stop and Uninstall		Stop	Update
STOPPING	~~Fail the Coordination (end up in ACTIVE)~~If stopping aborted, ACTIVE. Otherwise, error.		Wait for stop to complete then Uninstall.	Wait for Stop to complete then Start.		Wait for Stop to complete then Update.
UPDATING	~~Fail the Coordination (go back to previous state)~~If updating aborted, assume previous state. Otherwise, error.		Wait for Update to complete then Uninstall	Wait for Update to complete then Start.		Wait for Update to complete then Update.
UNINSTALLING	~~Fail the Coordination (end up in INSTALLED)~~If uninstalling aborted, INSTALLED. Otherwise, error.	Error		Error	Error	Error
UNINSTALLED	Error	Error		Error	Error	Error

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

8.1 State Tables: Interruptable operations

Note: This approach was considered and rejected in favor of the simpler option outlined in section 7.1.

The following approach treats interrupts to transitional states as the norm. These interrupts are performed through the use of the Coordinator:

6. Transitional states are interrupted via Coordination. This can be done in one of two ways:
    1. A resource processor detects a problem and fails the Coordination.
    2. An admin agent chooses to cancel an in-flight operation (e.g. cancelling a start operation whilst in the STARTING state).
7. If the subsystem is in a state beyond (needs defining) that which would be achieved by processing the operation, then it is considered a no-op (e.g. install on an ACTIVE subsystem does not cause it to go to INSTALLED).
8. If an operation is called during a transitional state, and the transitional state is before the state that would be reached then it runs to completion and then performs the requested operation (e.g. start on an INSTALLING subsystem completes the install and then tries to reach ACTIVE).
9. Errors are typically IllegalStateExceptions.
10. No-ops are shown with blank cells.

	cancel	install	uninstall	start	stop	update
CREATED?		Install	Error	Install, Resolve and Start	Error	Error
INSTALLING	Fail the Coordination (end up in CREATED state? - bundles in UNINSTALLED?)		Fail Coordination (end up in CREATED state?)	Complete Install, then do Resolve and Start		Error
INSTALLED			Uninstall	Resolve and Start resources		Update
RESOLVING			Uninstall	Complete Resolve then Start		Complete Resolve then do Update
RESOLVED			Uninstall	Start		Update
STARTING	Fail the Coordination (end up in RESOLVED)		Fail the Start Coordination and Uninstall		Fail the Start (end up in RESOLVED)	Fail the Start then do Update
ACTIVE			Stop then Uninstall		Stop	Update
STOPPING	Fail the Coordination (end up in ACTIVE)		Complete the Stop then Uninstall	Fail the Stop Coordination (end up in ACTIVE)		Complete the Stop then do Update
UPDATING	Fail the Coordination (end up in state prior to update).		Fail the Update then Uninstall	Complete Update then do Start		Complete the Update then do next Update
UNINSTALLING	Fail the Coordination (end up in INSTALLED)	Error		Error	Error	Error
UNINSTALLED		Error		Error	Error	Error

8.2 Metadata Formats

XML was considered for the format of the Subsystem and Deployment definitions but rejected because the power and associated complexity/verbosity of XML was not required to express the subsystem information.

Java properties files were also considered for the format of the Subsystem and Deployment definitions but these suffered from the opposite problem from XML. Whilst Java properties files can use the same colon separator as Java manifests for name-value pairs and would therefore be familiar to OSGi developers, they do not have the concept of attributes and are therefore too simplistic to easily represent a Subsystem or Deployment definition.

8.3 Zip versus Jar

This design proposes the use of a zip file for the subsystem archive format. A design which used a jar format was considered for the following reason:

1. to enable subsystems to be installed as bundles using the extender pattern.
2. To enable the artefact to be signed.
3. To enable JarInputStream to be used to load them without needing to test first whether or not it was a zip.

After investigation the following conlusions were reached. Regarding 1, whist it seems attractive to manage a subsystem as a bundle, this would lead to strange lifecycle bundles living in the runtime. Also, these bundles would not be able to replace the need for a SubsystemAdmin services and therefore their value is limited. Regarding 2, the current belief is it is sufficient to sign the bundles contained within the archive, not the archive itself. If this is found not to be sufficient, then this discussion can be revisited. Regarding 3, it is simple to load a zip file using a JarInputStream and therefore this is a non-issue.

One concern that stems from the use of the jar format is the need to have a META-INF/MANIFEST.MF that serves no purpose other than to potentially confuse the developer.

8.4 Resource Processors

Resource processor design is inspired by deployment admin. A resource processor is a service that adds the ability to manage the life-cycle of resources on behalf of a subsystem. For example, a resource processor might manage subsystem configuration, or certificates.

8.4.1 Resource Processor Services

Each resource processor is registered in the service registry under the interface `org.osgi.service.resource.ResourceProcessor`. The types of resources a resource processor can handle are identified using the service property **osgi.resource.namespace**. The namespace values are the same as those used in the generic requirements and capabilities (RFC 154) and OBR (RFC 112), or should follow the best practice for defining custom namespaces. The type of this property is String+ to allow a single resource processor to provide support for multiple namespaces.

The Subsystem implementation must only use resource processor services that are visible to its implementation bundle. In other words the service must be registered in the *root scope*. The subsystem implementation must not look inside other scopes to find resource processors. Note, if a subsystem exports a resource processor up to the root scope then this resource processor becomes a candidate to be used by the subsystem runtime.

If there are multiple resource processors that support the same namespace, then the one returned by **getServiceReference** (i.e. highest ranking, lowest id (tie breaker)) must be used.

TBD: Describe how to 'observe' resource life-cycle without being 'the' resource processor (Resource Listener).

8.4.2 Resource Processor Life-cycle

TBD: Define what happens when you remove/replace a resource processor. Do we take down any dependent subsystems? Do we have a "forced" capability similar to that used by deployment admin?

8.4.3 Resource Processor Operations

Resource processors implement a **process** operation that is called for each of the life-cycle operations that can be performed on a subsystem itself (install, uninstall, start, stop and update). The **process** operation is passed a ResourceOperation that describes the following:

- The operation to be performed.
- The context of the operation (this can include a reference to the subsystem to which the resource belongs.
- The coordination under which the operation is being performed. If no coordination is provided, then the operation is performed independent of any others. If a coordination is provided then the resource processor must register as a participant and must fail the coordination if it fails to process the resource.

The subsystem runtime uses the coordinator service to coordinate the outcomes of these operations across resource processors for a single subsystem. The resource processor must process each request immediately and only return once the processing has completed. If a resource processor is notified that the coordination **failed** then it must cancel or attempt to undo any actions requested under that coordination. If the resource processor cannot undo all actions then it must throw an exception from the **Particpant.failed** call.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10Document Support

10.1 References

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

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

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	Guillaume Nodet
Company	Progress
Address	
Voice	
e-mail	gnodet@progress.com

Name	Graham Charters
Company	IBM
Address	
Voice	
e-mail	charters@uk.ibm.com

10.3 Acronyms and Abbreviations

10.4 End of Document