131 Resolver Service Specification

Version 1.0

131.1 Introduction

Today very few applications are self contained, the prevalent development model is that applications are built from (external) components, often open source. Application developers add business logic and glue code and assemble the diverse components into a resource that provides the desired capabilities when deployed. Designing the assembly has long been a manual, error prone, process. Although the direct dependencies are given, the largest number of dependencies are usually the transitive dependencies: the dependencies of the dependencies. Modern applications can end up with hundreds to thousands of external dependencies. Numbers that make tooling inevitable.

The OSGi framework is the first specification that provides a foundation for automating a significant part of this assembly process. The Requirement Capability model from the [1] *Core Specification 4.3* provides a dependency model that allows a resource to express its dependencies, constraints and capabilities. If a resource's constraints are met it provides capabilities that can satisfy other resource's requirements. The OSGi dependency model is fully generic and is not limited to bundles. Resources can be bundles but also certificates, plugged in devices, etc.

Resolving the transitive dependencies is a non-trivial process that requires careful design to achieve the required performance since the underlying problems is NP-complete. OSGi frameworks have always included such resolvers but these were built-in to the frameworks and not usable outside the framework for tooling, for example automatically finding the dependencies of a bundle that needs to be installed.

Tooling automating dependency management is becoming more and more important in today's computing environments since the number of nodes that must be provisioned as well as the the number of dependencies is rapidly increasing a threshold where manual methods no longer can provide reliable results.

This specification therefore provides the Resolver service, a service that can be the base for provisioning, deployment, build, and diagnostic tooling. The service can take a requirement and resolve it to a wiring of a set of resources.

For example, with cloud computing a new requirement can be translated into a new OSGi Framework instance being started on a node and provisioned with the set of bundles that satisfy the given requirement. The OSGi Resolver service would be a corner stone of such an auto-provisioning tool.

However, the OSGi Resolver service is not limited to these higher end schemes. Build tools can use the Resolver to find components for the build path and/or run time environment and predict the results of installing a set of bundles on a target environment.

The OSGi Resolver service is an essential part of a software model where applications are build out of independent components.

131.1.1 Essentials

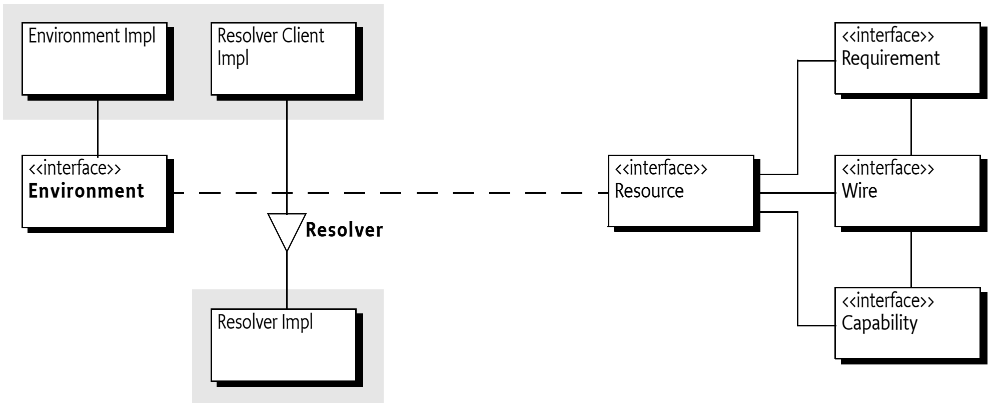
- *Transitive* – From a requirement, find a consistent set of resources that satisfy that requirement.
- *Diagnostics* – Provide diagnostic information when no resolution can be found.
- *Scoped Repositories* – Allow the callers to control the repository view.
- Build Tools – Must be useful in establishing build and and runtime class paths.
- *Provisioning* – Must be useful to find a set of bundle that can be installed in a system without running into unresolved problems.

- *OSGi* – Resolvers must provide the semantics of all the OSGi name spaces as well as the uses constraints.
- *API*– The API for the Resolver must provide the base for the Framework Bundle Wiring API.
- *Performant* – Enable highly performant implementations.
- *Frameworks* – Allow Frameworks to provide their resolver as a service.
- *Scalable* – Allow access to very large repositories.

131.1.2 Entities

- *Environment* – An interface implemented by the caller of the Resolver service to provide the context of the resolution.
- *Wiring* – The wiring, maps Resources to the wires that connect them to other Resources.
- *Resolver* – A service that can find Wirings that resolve all the (transitive) dependents of the given requirements.
- *Wire* – Links a Requirement to a Capability.
- *Resource* –An artifact with Requirements that need to be provisioned and resolved to provide its Capabilities.
- *Requirement* – A necessity for a given capability, expressed as a filter expression on a set of Capabilities in a given Name Space.
- *Capability* – A set of attributes and directives describing according to a Name Space what a Resource provides when resolved.
- *Name Space* – The type for a Capability and Requirement
- *Resolution* – The result of a resolving operation

Figure 131.1 Class and Service overview



131.1.3 Synopsis

The Resolver service provides a servce to find a complete and consistent set of transitive dependencies. Such a set can be used to install resources in the framework from local or remote repositories. To use the Resolver service the client must provide an Environment object, a collection of mandatory Resources and a list of optional Resources.

A resolution can be based on an existing wiring, for example the current Framework state. The Environment can provide this existing state.

The Resolver then must satisfy all the Requirements of the mandatory Resources. If such requirements cannot be satisfied by the Capabilities of any of the given Resources then the Resolver must ask the provider to find additional Capabilities for the unsatisfied Requirements. Capabilities are always associated with a Resource, which is subsequently associated with additional Requirements. The Resolver must find a set of Resources that include the given set of mandatory Resources, has no unsatisfied mandatory Requirements, and is self consistent.

The Requirement-Capability model is fully generic but provides special semantics through the use of name spaces. The Resolver must implement the standard OSGi namespaces as described in [1] *Core Specification 4.3*, including the uses constraints. However, part of the semantic of the OSGi name spaces are delegated the Environment. Singletons, matching priority, and the exact matching rules like mandatory attributes are the responsibility of the Environment.

Requirements have an effective directive that indicates in what situations the Requirement must be resolved. The Resolver queries the Environment to find out if the Requirement is effective for a given invocation. Only Requirements that are effective need to be satisfied.

Since Capabilities are implemented by Resources that have additional Requirements the process must create a set where all transitive Requirements are satisfied or fail with a Resolution Exception. This Exception can provide additional details why the resolution failed, if possible.

At thend of a successful resolution the Resolver returns a, map of Resource to Wires, providing a possible topology of Resources wired through their Requirements and Capabilities. This information can then be used to provision missing Resources or to provide diagnostic feedback.

131.2 Provisioning Agent

Provisioning is the process of providing a framework with the necessary resources to allow it to operate according to set goals. In OSGi terms, this consists of installing bundles and ensuring that the configuration is setup correctly. With OSGi, bundles explicitly describe their Capabilities and Requirements. This can range from Export-Package (a Capability) to a generic Require-Capability header.

OSGi Frameworks have a resolution process that ensures Requirements are met before a bundle is even allowed to provide code tot he shared space. As long as the Requirements are not met the Bundle remains in the INSTALLED state and is thus prohibit from contributing. Provisioning is made harder on OSGi Frameworks because it combines the resolving stage with the decision stage. The Resolver service separates these two states and this allows a third part, a management agent, to interact to handle more scenarios, better diagnostics, etc.

For example, a management agent could use a Resolver service to find missing dependencies and install them on demand from a local directory with bundles. Such a Provisioning Agent could have the following skeleton:

```
@Component(service={ProvisioningAgent.class})
public ProvisioningAgent implements Environment {
  File                    bundles = ...;
  Map<String,Resource>    resources = getResources(bundles);
  Resolver                resolver = ...;
  BundleContext           context = ...;

  public void install( String location ) {
    Resource r = resources.get(name);
    if ( r == null) error(...);

    try {
      Set<Resource> provision = resolver.resolve(
        this,
        Arrays.asList(r),null).keySet();


      for (Resource rb : provision ) {
        String location = getLocation(rb);
```

```
            Bundle bundle = context.getBundle(location);
            if ( bundle == null)
               b = context.installBundle(location);
            bundle.start();
          }
        } catch(ResolutionException re) {
          ... // diagnostics
        } catch(BundleException re) {
          ... // diagnostics
        }
      }
      ... // Other methods
    }
```

### mention that the mandatory/optional resource is included in the search

131.2.1 Finding Capabilities

They key object that needs to be provided by the Provisioning Agent example is the Environment object. This object acts as the context for the resolving operation. Implementing an actual Environment object requires access to a set of Resources, the existing wiring, and it must decide about the effect of Requirements. In this example, the existing wiring and effective time are effectively ignored. This means that two methods must be implemented:

- findProviders(Requirement) – Return a sorted set of Capabilities matching the given Requirement.
- findProviders(Collection) – A batch version of the previous method so that multiple Requirements can be queried in one method.

In this example, only the singular method is shown since the batch method can be easily constructed from the singular version:

```
public SortedSet<Capability> findProviders(Requirement requirement) {
  SortedSet<Capability> result = new TreeSet<Capability>(DUMMY_COMP);
  for ( Resource r : resources )
    for ( Capability c : r.getCapabilities() )
      if ( match(requirement, c) )
         result.add(c);
  return result;
}
```

### discuss that preference should be wiring state, mandatory, optional

The result of the findProviders(Requirement) method is a Sorted Set. In the case the Provisioning Agent prefers certain Capabilities over others it can order them in the result. In this example a dummy comparator is used; this comparator always returns 0 since in this example there is no preference. In a real implementation it is likely that certain resources are more expensive to install than others. For example, resources could be remote, larger, cost money, etc.

### why do you need an expensive SortedSet for this? List should be ok?

Matching a Requirement to a Capability must be done by the Agent, the Resolver never matches Requirements to Capabilities. Matching is non-trivial and requires understanding the name space. For example, matching an Import-Package Requirement to an Export-Package Capability requires that mandatory attributes are asserted used in the match. For example:

```
boolean match(Requirement r, Capability c) {
  if ( !r.getNamespace().equals(c.getNamespace()))
    return false;
    Filter filter = context.createFilter(r.getDirectives.get("filter"));
    if ( !f.matches(c.getAttributes())
      return false;

    if ( "osgi.wiring.package".equals(c.getNamespace()) {
      String mandatory = c.getDirectives().get("mandatory");
      if ( mandatory == null)
        return true;

      List<String> attrs = Arrays.asList( mandatory.
            toLowerCase().split("\\s*,\\s*");

      Matcher m = Pattern.compile(
           "\\(\\s*([^=><~()]+)\\s*(=|~=|>=|<=)((\\\\.|[^\\)])*)\\)")
           .matcher( f.toString().toLowerCase() );

     while( m.find() ) {
       attrs.remove(m.group(1)); // the attribute name

      return mandatory.isEmpty();
    } else if ( ... )
      // other special name space rules

    return true;
  }
```

### mention fragments are treated normally

### discuss that the mandatory + optional resources are included in the search

131.2.2 Repositories

Resolving to provision a Framework is different than Framework resolving itself. During provisioning remote Repositories can be consulted. These Repositories generally contain magnitudes more bundles than what is installed in a framework. This specification allows a resolution to be influenced in such a way that it is appropriate for the current context, where the context could be a local Framework, a remote Repository, or something else. The purpose of the Repository is to allow a client of the Resolver to discover resources to be installed.

The previous example worked out some of the details of finding Capabilties from a Requirement. In general this work will be done by a Repository. The *Repository Service Specification* on page 529 provides the details for a service that abstracts a Repository that is Requirement/Capability aware. With a Repository service the findProviders(Requirement) method can be implement as follows.

```
List<Repository> repositories = new CopyOnWriteArrayList();
@Reference(cardinality=MULTIPLE)
void setRepository(Repository repository) {
  repositories.add(repository);
}
void unsetRepository(Repository repository) {
  repositories.remove(repository);
}
public SortedSet<Capability> findProviders(Requirement requirement) {
  SortedSet<Capability> result = new TreeSet<Capability>(DUMMY_COMP);
  for ( Repository repository : repositories )
      result.addAll( repository.findProviders(requirement) );
  return result;
}
```

131.2.3 Existing State

The Resolver service can take an existing state into account. For this reason, the Environment interface specifies the getWirings() method. This method returns a map of Resource to Wiring. A Wiring is an object that describes the actively wired state of a Resource. For example, it has no unwired Capabilities and Requirements and it handles the situation where Resources and Capabilities come from another Resource, Fragments require this distinction.

The Framework provides access to the existing state through the Framework Wiring API described in [2] *Framework Wiring API (Core)*. The Resolver service API is based on the generic Requirement Capability model that is defined in the Core; its API is in the org.osgi.framework.resource package. The actual Framework wiring contains additional details, see for example *Fragments* on page 7, and has therefore a corresponding interfaces for this package in the org.osgi.framework.wiring package. However, these interfaces extend their counterpart in the resource package. This design makes it trivial to include the current state as is demonstrated in the following example of the getWirings() method.

```
public Map<Resource,Wiring> getWirings() {
   Map<Resource,Wiring> wirings = new HashMap<Resource,Wiring>();

    for ( Bundle b : context.getBundles() ) {
      BundleRevisions revisions = b.adapt(BundleRevisions.class);
      if ( revisions != null )
         for ( BundleRevision revision : revisions.getRevisions() )
```

### or is it better to only use current wiring?

```
            wirings.put(revision,revision.getWiring());
    }

    return wirings;
}
```

### mention that if existing state is used that it in general should be found in findProviders

131.2.4 Effective

When a Resolver service is called there is a specific purpose. However, not all the Requirements have to be related to this purpose. For example, certain Requirements are only intended for the OSGi Resolving phase but other Requirements are meant to be used with the more dynamic services. In the Requirement-Capability model this is modeled with the effective directive. This directive has one defined value: resolve. This single defined value indicates that the Requirement should be satisfied in an OSGi Resolve operation. However, the directive can contain other values.

The Resolver is agnostic for this directive, it will always ask the Environment if a Requirement is effective or not with the isEffective(Requirement) method. The Environment can therefore use the Resolver service to in different stages. To make Requirements effective during the resolving of bundles it will be necessary to implement the isEffective(Requirement) method similar to:

```
public boolean isEffective(Requirement requirement) {
    return "resolve".equals(requirement.getDirectives().get("effective"));
}
```

131.2.5 Fragments

The Requirement-Capability model required additional complexity to handle Fragments. A Fragment is a very special Resource in OSGi because its Capabilities (for example an Export-Package) is exposed through its host. To model this asymmetry it was necessary to distinguish between the potential Requirements and Capabilities as they are available through the Resource, and the actual Requirements and Capabilities as they are available in the Framework.

The Resolver handles Fragments by synthesizing Requirements and Capabilities so that it can associate them with the host bundles. However, an Environment must ignore this additional handling, it must always treat a Fragment as a normal Resource, no extra matching logic is required for Fragments.

### are Fragments optional resources in Resolve method? How are they found?

131.2.6 Singleton Capabilities

A Bundle can be marked as a singleton bundle with the Bundle-SymbolicName header. Such bundles require that they have no visibility nor access to other bundles with the same symbolic name. This constraint is not enforced by the Resolver service, it is the responsibility of the client. In general this means that the client must not return Capabilities in a session that could select singleton Resources. However, this means that the resolution can fail while it might have succeeded when another candiate for the singleton had been selected. The simplest, though not most efficient, solution is to backtrack and select other candidates.

### need some tips what they should do with singletons

131.2.7 Diagnostics

The Resolver service throws a ResolutionException when the resolve operation cannot find a solution. This Exception provides the standard human readable message. However, there is also the getUnresolvedRequirements() method. With this method it is possible to find what Requirements could not be matched. Though this is very useful in many cases it must be realized that the resolving is a complicated process. It is not possible to establish the exact set of missing Requirements because any unresolved resolution can have many, different, sets of unresolved Requirements. This is an intrinsic part of the resolution problem. There is also no guarantee that providing Capabilities that satisfy these Requirements will give a successful resolution. There could still be other constraints that cannot be satisfied.

That said, the getUnresolvedRequirements() can often point to a potential solution.

131.2.8 Installing a Wiring

###

131.3 Another Resolver Use Case

The primary use case for the Resolver is in relation to the OSGi framework: deployment, provisioning, and diagnostics. However, the Resolver service is completely generic and can be used for other purposes. For example, the popular [3] *JSR 330: Dependency Injection Framework* when used in OSGi would require a non-trivial resolving phase if the injection targets and object providers could come from different bundles, especially considering the uses constraints. This example shows how the OSGi Requirement Capability model can be used to implement a toy injection framework. Since it is an example it only supports field injections, only singleton scope (type name is instance name), and does not handle any errors.

The example should be able to wire up the following type of code:

```
public class Target {
  @Inject Interface field;
}
...
public class Implementation implements Interface {}
```

Any model based on the OSGi Requirement-Capability model requires a Name Space. For this example, the following Name Space is chosen:

```
com.toy.inject
```

JSR 330 supports qualifiers. The @Named annotation is used to put a name qualification on an ibjection site; a provider must provide this name. This trivially mapes to an attribute with a fixed name: named. The @Qualifier annotation goes further, it targets other annotations and makes them qualifiers. These targeted annotations can have elements. Also these qualifiers can be translated to an attribute by appending the fully qualified name of the qualifier annotation with the name of the annotation element. For example:

```
@Qualifier
@interface Color {
   String value();
}
```

The attribute name for this annotation then becomes:

```
com.example.Color.value
```

The following example shows how the qualifiers can be used on an injection site:

```
@Inject @Color("brown") @Named("left")
Interface field;
```

The example code can then be translated into the following Requirement filter:

```
(&(type=com.example.Interface)(com.example.Color.value=brown)(named=left))
```

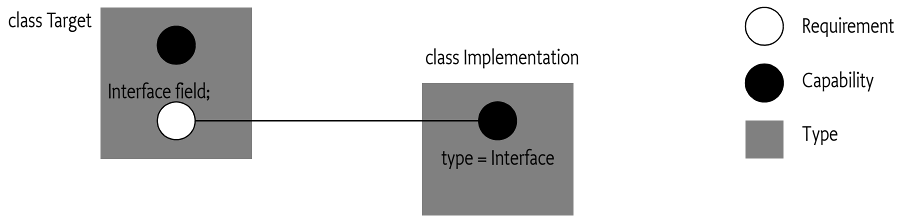
A Name Space requires the definition of its attributes, this is done in Table 131.1 for this example Name Space.

Table 131.1 *com.toy.inject definition*

Attribute Name	Type	Syntax	Description
type	String	Fully Qualified Name	The fully qualified name of the requested type.
impl	String	Fully Qualified Name	The name of the implementation class.
*	String		Each qualifier should be expanded with its property names. For example, com.foo.Xyz.bar for qualifier annotation @Xyz(bar=3)
named	String		The name in a @Named qualifier annotation

In this model, each singleton type available for injection is defined by a capability. This singleton type then has 0 or more Requirements on other singleton types; each of the requirements map to a field injection. For example, a type A could have a field b of type B and c of type C. The type A would be a capability provding a singleton instance A. The fields would map to two requirements. This model is depicted in Figure 131.2.

Figure 131.2 *Toy DI*



JSR 330 does not define annotations for the providers since this works differently for Guice and Spring, the authors of JSR 330. In [4] *Google Guice* the providers are defined in the Bindings of a Module while Spring uses XML. In this example, for simplicity, the metadata is provided with the Provide-Capability Manifest header. For example:

```
Provide-Capability:
  com.toy.inject;
  type=    com.example.Interface;
  impl=   com.example.Provider;
  uses:=   "com.example,com.bar,com.foo"
```

Given such a Capability it is possible to instantuate the singleton from the proper bundle since a Capability is always associated with a Bundle Revision and thus a bundle. This is also the reason the uses directive must be specified, this allows the Resolver service to create a wiring topology that does not violate class loading constraints.

A field injection maps to a Requirement. However, Requirements and Capabilities are related to a Resource, it is not immediately clear how to find out what Requirement belongs to what field injection. The solution is to use the Requirement's attributes. Such an attribute could indicate the implementation type and thus the singleton name along with the field to be injected. Finding the singleton to inject is then left to the Resolver service since this involves matching of the qualifier attributes as well as finding the interface type.

An attribute on this header could specify the injection site: the type name to be injected and the field, method, or constructor. The impl Requirement attribute will indicate the implementation class name (could also be a target.) The field Requirement attribute holds the name of the field to be injected.

A source of confusion is the difference between the matching attributes that are defined in the Name Space and the attributes associated with a Requirement. A Requirement is primarily a filter but has the possibility to carry additional attributes. The toy injector takes advantage of this feature by using it to specify the injection set with the field and impl Requirement attributes. For example:

```
Require-Capability:
  com.toy.inject;
  effective:=com.toy.inject;
  filter:=;"(type=com.example.Interface)";
  impl=com.example.Target;
  field=field
```

The Injector has the responsibility to wire the targets and implementors together. In this example, the Injector uses the something like the extender model to find applicable models, this is not further shown:

```
public class Injector implements Environment {
   Set<BundleResource>  resources  = ...;
   Resolver             resolver   = ...;
```

Next stage is resolving, this is almost ridicously simple since the Resources are already available:

```
   public void wire() throws Exception {
      Map<Resource, List<Wire>> result =
         resolver.resolve(this, resources, null);
      inject(result.values());
   }
```

Since the example ony resolves already resolved bundles it is not necessary to look at any Requirements but the Requirements modeling the field injection. The isEffective(Requirement) method can therefore be implemented as follows:

```
   public boolean isEffective(Requirement requirement) {
      return "com.toy.inject".equals(requirement.getNamespace());
   }
```

The Injector provides all the possible Resources to the resolve(Environment,Collection,Collection) method. It is therefore possible to provide dummies for the findProviders(Requirement) and findProviders(Collection) methods.

```
   public SortedSet<Capability> findProviders(Requirement requirement) {
      return new TreeSet<Capability>();
   }
   public Map<Requirement, SortedSet<Capability>> findProviders(
      return new Map<Requirement,SortedSet<Capability>>();
   }
```

If there is a resolution then the result local variable holds the answer: each Wire maps to a field injection. However, Dependency Injection requires that the injection is ordered. Odering requires an intermediate structure that represents the singleton before it is constructed. The Injection class in this example. It has the class of the singleton and it caches the instance value once it is created. Since it is necessary to inject the fields from its requirements the Injection class holds a map that maps the field name to another Injection object. When the instance is instantiated this map can be used to find the to be injected objects. During the injection phase it is necessary to track forward references since the order of the Wires is undefined.

```
   Map<String, Injection>injections= new HashMap<String, Injector.Injection>();
   class Injection {
      Class< ? >    type;
      Object        instance;
      Map<String, Injection>fields= new HashMap<String, Injection>();
      ...
   }
```

The inject method then must traverse all the wires and then for each wire administer the proper field injection. Each Wire has a Capability (an instance/type) and a Requirement (a field injection.) Both the Capability and the Requirement are mapped to an Injection object keyed on the impl attribute. The Capability's Bundle is then used to load the appropriate class. The Injection object that corresponds to the Requirement is then added to the fields map of the Capability's Injection object.

```
   void inject(Set<List<Wire>> wires ) throws Exception {
      for (List<Wire> lw : result.values())
         for (Wire w : lw) {
            Requirement r      = w.getRequirement();
            Capability c       = w.getCapability();

            String toClass     = (String) c.getAttributes().get("impl");
            String fromClass   = (String) r.getAttributes().get("impl");
            String field       = (String) r.getAttributes().get("field");

            Bundle bundle      = ((BundleRevision)c.getResource()).getBundle();
```

```
                Injection to      = getInjection(toClass);
                Injection from    = getInjection(fromClass);

                to.type           = bundle.loadClass(toClass);
                to.fields.put(field, from);
            }

        for (Injection i : injections.values())
            i.get();
    }
```

The get() method in the Injection class uses itself recursively to build the objects in dependency order:

```
        Object get() throws Exception {
            if (instance != null)
                return instance;
            instance = type.newInstance();
            for (Map.Entry<String, Injection> entry : fields.entrySet()) {
                Field field = type.getField(entry.getKey());
                field.set(instance, entry.getValue().get());
            }
            return instance;
        }
```

And last the getInjection method:

```
    Injection getInjection(String type) {
        Injection i = injections.get(type);
        if (i != null)
            return i;
        i = new Injection();
        injections.put(type, i);
        return i;
    }
```

131.3.1 Existing State

The toy Injector can support existing state and rewire the resources from an existing state. For this, the Injector must implement the getWirings() method. This requires implementing the Wiring interface. Since this is a non-trivial interface using an existing state is left as an exercise.

131.4 Resolver Service

The Resolver service is a stateless service that can be invoked to create *resolution*. A resolution consists of a set of Wires between Resources. If an existing set of Wires is provided then it is a delta on this existing set.

131.4.1 Resolver Model

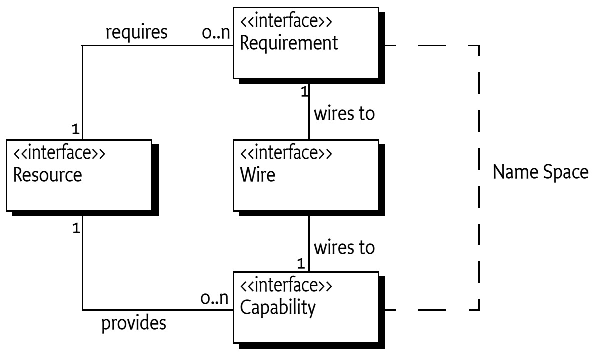
The *Resolver* service is an interface to a generic constraint solver based on the *Require-Capability* model defined in [2] *Framework Wiring API (Core)*. This model defines a constraint-solving language that is used by the Framework (org.osgi.framework.wiring) to create the mesh of Class Loaders but that has been designed to be useful in solving many other problems as well. For this reason, the Framework API provides the org.osgi.framework.resource package. This package contains a number of interfaces for the constraint solving language.

This model is based on a small set of entities, summarized as follows:

- *Resource* – Installable unit like a Bundle, certificate, etc. Provides Capabilities and has Requirements on other Capabilities.
- *Capability* – A set of attributes and directives that describe a feature of the Resource. A Capability can only satisfy a Requirement in the same Name Space.
- *Requirement* – Expresses the need for a Capability that satisfies the Requirement's filter. A Resource can only provide its Capabilities if all its Requirements are satisfied.
- *Name Space* – Provides a schema name for the attributes of the Capability. A Capability can only match a Requirement if it is in the same Name Space. The Name Space can also add additional constraints outside the generic model.
- *Wire* – A link between a Requirement and a matching Capability.

The entities and their relations are depicted in Figure 131.3 on page 12.

Figure 131.3 org.osgi.framework.resource (simplified)



The task of the Resolver is to accept a set of Resource and for these Resources create a Wire for each mandatory Requirement that *wires* to a matching Capability.

131.4.2 Resolving

The Resolver service must find a set of Wires that connect Resources from their Requirements to Capabilities. Though there are many solutions possible, it must find a solution that fullfils the requirements outlined in this section. For a legend of the mathematical symbols, see ###.

### I need some help from Glyn for the equations, I think they contain intent now but need formalization ...

131.4.2.1 Variables

The resolve(Environment,Collection,Collection) method has the following parameters:

E	:	Environment	Controls the resolution
R_m	:	□□Resource	Immutable set of mandatory Resources
R_o	:	□□Resource	Immutable set of optional Resources

During resolving the Resolver service can callback E to retrieve the following information:

C_{env}	:	□□Capability	Answers from the findProviders methods
Q_{eff}	:	Requirement	Set of effective Requirements: isEffective(Requirement)
X	:	Resource□□ □Wiring	An existing Wiring state, getWirings()

### Glyn, □ □is ok for this?

The resolve method returns any additional wiring that is necessary to add the mandatory Resources on top of the existing wiring X:

D	:	Resource□□ □Wire	A Proposed wiring.

X and *D* are actually of different types. The getWirings() and resolve(Environment,Collection, Collection) method return:

```
Map<Resource,Wiring>        getWirings()
Map<Resource,List<Wire>>    resolve(...)
```

To simplify the assertions it is easier to treat them as the same type since *D* is a delta on top of *X*. Both *D* and *X* can trivially be mapped to a Resource □ Wire tuple set (Resource□Wire). Therefore, the following variable *W* (all wires in the solution) and *R* (all Resources in the solution) are introduced:

```
W :   □Wire =
      w : X Wiring • w.providedResourceWires
   □  w : X Wiring • w.requiredResourceWires
   □  D□Wire

R :   □Resource =
      XResource
   □  DResource
```

131.4.2.2 Sanity Checks

The following checks are sanity checks to make sure the wiring is consistent and follows the basic rules. That is, wires must only refer to Resources in the solution (R), and Requirements and Capabilities must only come from Resources in R.

```
□ w  :  W
  •  w.provider □ R
  □  w.requirer □ R
  □  w.capability.resource □ R
  □  w.requirement.resource □ R
```

It is also mandatory that a wire only links Requirements and Capabilities in the same Name Space.

```
□ w  :  W
  •  w.requirement.namespace□□ □w .capability.namespace
```

131.4.2.3 Consistency of Resolution and Existing State Relations

Since *D* and *X* are redundant (the Wires refer to the Resource and vice-versa) it is necessary that the resolver does not introduce errors. A wire must come from either *X* (the existing state) or *D* (the delta state) and not from both. That is, the Resolver must not add duplicate Wires.

```
□ w  :  W
  •  □w.provider □  w□□□D      □     w.provider □  w  □ X□
  □  □w.requirer □  w□□□D      □     w.requirer □  w  □ X□□
```

131.4.2.4 Effective

Only *effective* Requirements are part of the solution. The effectiveness is decided by the isEffective(Requirement) method from the Environment and Q_e contains the effective Requirements.

```
□ w  :  W
  •  w.requirement □□□Q_eff
```

131.4.2.5 Matching

The Requirement of a Wire must *match* the corresponding Capability. Though the Resolver must be aware of the OSGi Name Spaces, most of the knowledge in this area resides with the Environment. The Resolver finds out that a Requirement matches a Capability by calling either the findProviders(Requirement) or findProviders(Collection) method. Either method returns a mapping from one Requirement or a number of Requirements to a number of Capabilities, the variable *M* holds these tuples. The resolver therefore never matches Requirements to Capabilities, it uses the answers from the Environment. The answers from the Environment .

An implication of this is that a Resolver queries the Environment for of the Requirements found on the mandatory and optional Resources given in the invocation, as well as the existing Wiring state *X*. The Resolver has no means to detect that these Resources could also satisfy the Requirements and will therefore not automatically do so.

Fragments require special handling. In a Framework resolution the Requirements and Capabilities of a Fragment are used to wire the host, the Fragment is never wired except for its host requirement. However, an Environment must treat a Fragment Bundle Resource as any Resource and return the Capabilities from the Fragment. The Resolver service will then Synthesize Requirements and Capabilities for the host, see *Synthesized Resources, Requirements and Capabilities* on page 15.

Matching rules are further discussed ###.

The matching rules must follow all OSGi name space rules.

### more about matching, what can we assert here?

131.4.2.6 No New Resources

A Resolver is not allowed to add additional Resources that were not provided by the caller. Only Resources given in the resolve(Environment,Collection,Collection) method or provided through the Capabilities returned from one of the findProviders methods, or Resources from the getWiring method may be included:

$\square\ r : R$

$\bullet\ r \square\square\square R_m$

$\square\ r \square\square\square R_o$

$\square\ r \square\square\square X\ Resource$

$\square\ r \square\square\square\square c : C_{env}\ \bullet\ c\ resource\ \square\square$

131.4.2.7 Uses Constraints

Any solution must obey the *uses constraints* defined in [1] *Core Specification 4.3*. Uses constraints ensure that bundles cannot view different class loaders for the same package. That is, uses constraints provide the metadat to allow the Resolver service to make sure that the class space is consistent. Uses constraints are primarily constraints on package exports, importing such a package requires that the import of the packages mentioned in the uses constraints come from the same source. However, any Capability can specify uses constraints. The semantics of those uses constraints are that

$\square\ w : W;\ u : w.capability.uses$

$\bullet\ package(w.provider, u)\ \square\ package(w.requirer, u)$

$\square\ package(w.provider, u) = \square$

$\square\ package(w.requirer, u) = \square$

The function *package(R,Package)* establishes the provider of a package for a given requirer:

```
package( r : R, p : Package) =□□
    w : W
        □ w.requirer                                      □ r
        □ w.requirement.namespace                         □ osgi.wiring.package
        □ w.capability.attributes.get(osgi.wiring.package) □ p
        • w.provider□
□
```

131.4.2.8 Mandatory Requirements

A correct solution must also provide a wire for all mandatory wires:

$\square\ q : \square r : R \bullet r\ requirements \square\square$

$\square\ q\ resolution\ \square\ \text{mandatory}$

$\bullet\ \square\ w : W \bullet w\ requirement \square\square\square q$

131.4.2.9 Mandatory Resources

Last, but not least, the Resolver must ensure that each mandatory Resource is included in the solution:

$$R_m \subseteq R$$

131.4.3 Synthesized Resources, Requirements and Capabilities

Fragments can be attached to multiple hosts in an OSGi framework. Except for their osgi.wiring.host Requirement, all their Requirements and Capabilities are provided through the host. In practice, it is necessary for the Resolver to synthesize Requirements and Capabilities to track the host they correspond to since this differs from the originating Resource. These synthesized objects indicate that their ###

### there is no useful information in the RFC nor the initial spec draft about Synthesized.

Are all objects synthesized? Need some explanation ...

Dont understand why Resource is ever synthesized?

131.4.4 Resolution Exception

If the Resolver cannot find a solution or it runs into problems then it must throw a Resolution Exception, which is a Runtime Exception.

The ResolutionException provides the getUnresolvedRequirements() method. If the resolution failed then it is posisble that this was caused because it failed to find matches for certain Requirements. The information in this method can be very helpful find a solution that will work, however, there are a number of caveats.

Resolving is an NP-complete problem.For these problems is that there is no algorithm that can infer a solution from the desired outcome. Therefore, the Resolver tries a solution and if that solution it will backtrack and attempt another solution. An unavoidable aspect of such solutions is that it is impossible to pin-point a single failure point if the algorithm fails to find a solution, in general the algorithm gives up after having exhausted its search space. However, during its search it might have been very close to a solution, for example it only missed a single requirement, but its final failure missed many requirements.

The implication is that the reported missing Requirements neither give a guarantee for a resolution when satisfied nor indicate that this is the smallest set of missing Requirements.

In addition, the caller is in charge of the Environment that provides the Capabilities for any Requirements that the Resolver discovers. If this Environment could not find any providers for this Requirement it should be unlikely that any other automated mechanism could.

Therefore, getUnresolvedRequirements() is intended for human consumption and not for automated solutions.

131.5 Environment

The Resolver is intentionally kept stateless, it does not remember state, it does it retrieve its information about the environment through a service model, nor has it a hidden connection to the current Framework. The resolve method can be regarded as a pure functon without side effects. As input it receives the sets of mandatory and optional Resources and the Environment object. The Environment provides the context for a resolution. The Environment interface is implemented by the client of the Resolver service; it provides the context for a resolution operation.

Implementing an Environment interface is a non-trivial amount of work and requires a thorough understanding of the OSGi Framework resolution process. In general, the Environment interface is implemented by the invoker of the resolve operation since there is an interaction between the operation's other parameters and the implementation of the Environment's methods.

### we should consider making the Environment a concrete class so that it can be evolved

The Environment has the following responsibilities:

- Provide Capabilities that match one or more effective Requirements on demand.
- Decide if a Requirement is effective for the resolution.
- Deliver the existing state on which the resolution should be built.

These responsibilities are further defined in the subsequent sections.

131.5.1 Providing Capabilities

The Environment is responsible for

To find providers of a requirement:

- SortedSet<Capability> findProviders(Requirement requirement)

This API returns the capabilities within the environment that can satisfy the requirement. Capabilities at the beginning of the set are preferred over ones at the end. An environment implementation can choose to return a set that is lazily populated, preferring local capabilities over remote ones available in a repository.

The returned `SortedSet` must be mutable by the Resolver as the Resolver potentially modifies it with capabilities provided through fragments when the resolver attaches them to hosts. The capabilities added by the resolver always implement the `Synthesized` interface. This interface provides a `Synthesized.getOriginal()` API which provides access to the original fragment capability. The Environment must position these added capabilities appropriately in the `SortedSet`.

The Environment also provides an API to find providers of a List of Requirements:

```
Map<Requirement, SortedSet<Capability>>
 findProviders(Collection<Requirement> requirements)
```

To find the existing wirings which needs to be taken into account when resolving use:

```
Map<Resource, Wiring> getWirings()
```

The entity invoking the Resolver must provide the Environment to interact with. Example environment implementations can be found below.

Example Environments

A simple environment working with repositories for provisioning into an empty framework:

```
public class ResourceTrackingEnvironment implements Environment {
 private final Repository[] repositories;
 private final HashMap<Resource, Repository> tracked =
  new HashMap<Resource, Repository>();

 public ResourceTrackingEnvironment(Repository[] repositories) {
  this.repositories = repositories;
 }

 @Override
 public SortedSet<Capability> findProviders(Requirement requirement) {
  SortedSet<Capability> providers = new TreeSet<Capability>();
  for (Repository rep : repositories) {
   Collection<Capability> found = rep.findProviders(requirement);
   for (Capability cap : found) {
    Resource res = cap.getResource();
    tracked.put(res, rep);
    providers.add(cap);
   }
  }
  return providers;
 }

 @Override
 public Map<Requirement, SortedSet<Capability>> findProviders(Collection<Requirement> require-
ment) {
  // needs to be provided.
 }

 public Repository getRepository(Resource res) {
  return tracked.get(res);
 }

 @Override
  public Map<Resource, Wiring> getWirings() {
  return Collections.emptyMap();
 }

 @Override
  public boolean isEffective(Requirement requirement) {
  return true;
 }
}
```

An environment to represent the current framework state may look like this:

```
public class FrameworkEnvironment implements Environment {
 private final BundleContext bundleContext;

 public FrameworkEnvironment(BundleContext ctx) {
  bundleContext = ctx;
 }

 @Override
 public Collection<Capability> findProviders(Requirement requirement) {
  Set<Capability> capabilities = new HashSet<Capability>();
  for (Bundle bundle : bundleContext.getBundles()) {
   BundleRevision rev = bundle.adapt(BundleRevision.class);
   if (rev != null) {
    for (Capability cap : rev.getCapabilities(requirement.getNamespace())) {
     if ( ResolverUtil.matches(cap, req)) {
      capabilities.add(cap);
           }
          }
   }
  }
  return capabilities;
 }

 @Override
 public Map<Resource, Wiring> getWiring() {
  Map<Resource, Wiring> wiring = new HashMap<Resource, Wiring>();
  for (Bundle bundle : bundleContext.getBundles()) {
   BundleRevision revision = bundle.adapt(BundleRevision.class);
   if (revision != null)
    wiring.put(revision, revision.getWiring());
  }

  return wiring;
 }

 @Override
 public boolean isEffective(Requirement requirement) {
  return true;
 }
}
```

Resolver

Resolving is a complex constraint-processing operation which, given a set of resources and an environment produces a set of resources and wires between them that need to be created to apply the resolution. A resolver can be used for a number of purposes. It can be used by the framework itself to obtain the wires that need to be created to resolve a bundle or set of bundles. A resolver can be used with a remote repository to establish the transitive set of resources (bundles and possibly other artifacts) needed to successfully provision a root set of bundles into a framework. Additionally, a resolver can be used to analyze the impact of installing a set of resources into a framework.

A resolver is a stateless Service with the following API

Map<Resource, List<Wire>> resolve(Environment environment,

Collection<? extends Resource> mandatoryResources,

Collection<? extends Resource> optionalResources)

The resolve operation takes an environment which is used to find capabilities for requirements the resolver needs satisfied. Additionally the environment is used to retrieve the wiring that must be taken into account when computing the resolution.

The resolve operation also takes collections of resources to resolve. The mandatory resources must be resolved, failure to do so causes a `ResolutionException`. Optional resources are attempted to be resolved, but failure to do so does not cause an exception.

The resolve operation returns a map of resources with associated wires. This map represents the delta between the wiring provided by the environment and the resolved result. The map of resources may contain resources that need to be downloaded from a repository if the environment has the ability to consult repositories.

Resolver examples

Provision a resource and its dependencies into the framework.

```
void provision(Resource resource, Environment env) {
 Resolver resolver = getResolver(); // from Service Registry
 try {
  Map<Resource, List<Wire>> delta = resolver.resolve(env,
   Collections.singleton(resource), null);

  for (Resource r : delta.keySet()) {
   if (isBundleOrFragment(r)) {
    bundleContext .installBundle(getBundleID(),
            r.getContent());
   }
  }
 } catch(ResolutionException re){
      log.log("Unable to resolve " + resource +
       " Caused by: "  + re.getUnresolvedRequirements());
     }
    }

  boolean  isBundleOrFragment(Resource r) {
   Object type =
    r.getCapabilities(ResourceConstants.IDENTITY_NAMESPACE).iterator().next().
     getAttributes().get(ResourceConstants.IDENTITY_TYPE_ATTRIBUTE);

   return  ResourceConstants.IDENTITY_TYPE_BUNDLE.equals(type) ||
      ResourceConstants.IDENTITY_TYPE_FRAGMENT.equals(type);
  }

  String getBundleID(Resource r) {
   Object id =
    r.getCapabilities(ResourceConstants.IDENTITY_NAMESPACE).iterator().next().
     getAttributes().get(ResourceConstants.IDENTITY_NAMESPACE);
   if (id instanceof String)
    return  (String) id;
   else
    return null;
  }
```

More examples...

131.6 org.osgi.service.resolver

Resolver 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.resolver; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

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

131.6.1 Summary

- Environment – An environment provides options and constraints to the potential solution of a Resolver.resolve(Environment, Collection, Collection) operation.
- ResolutionException – Indicates failure to resolve a set of requirements.
- Resolver – A resolver is a service interface that can be used to find resolutions for specified resources based on a supplied Environment.
- Synthesized – During the resolution process it is necessary for the resolver to synthesize resources to represent the merge of bundles and fragments.

131.6.2 Permissions

131.6.3 public interface Environment

An environment provides options and constraints to the potential solution of a Resolver.resolve(Environment, Collection, Collection) operation.

Environments:

- Provide capabilities that the Resolver can use to satisfy requirements via the findProviders(Requirement) method
- Constrain solutions via the getWirings() method. A wiring consists of a map of existing resources to wires.
- Filter transitive requirements that are brought in as part of a resolve operation via the isEffective(Requirement).

An environment may be used to provide capabilities via local resources and/or remote repositories.

A resolver may call the findProviders(Requirement), isEffective(Requirement) and getWirings() method any number of times during a resolve using any thread. Environments may also be shared between several resolvers. As such implementors should ensure that this class is properly synchronized.

Concurrency Thread-safe

131.6.3.1 public SortedSet<Capability> findProviders (Requirement requirement)

requirement the requirement that a resolver is attempting to satisfy

□ Find any capabilities that match the supplied requirement.

A resolver should use the iteration order or the returned capability collection to infer preference in the case where multiple capabilities match a requirement. Capabilities at the start of the iteration are implied to be preferred over capabilities at the end.

The set returned by this call should be mutable to support ordering of Synthesized resources created by the resolution process.

Matching

A capability matches a requirement when all of the following are true:

- The specified capability has the same name space as the requirement.
- The filter specified by the filter directive of the requirement matches the attributes of the specified capability.
- The standard capability directives that influence matching and that apply to the name space are satisfied. See the capability mandatory directive.

Returns an collection of capabilities that match the supplied requirement

Throws NullPointerException – if the requirement is null

131.6.3.2 public Map<Requirement,SortedSet<Capability>> findProviders (Collection<? extends

Requirement> requirements)

requirements the requirements that should be matched

☐ Find any capabilities that match the supplied requirement.

The set returned by this call should be mutable to support ordering of `Synthesized` resources created by the resolution process.

See `findProviders` for a discussion on matching.

Returns A map of requirements to capabilities that match the supplied requirements

Throws `NullPointerException` – if requirements is null

See Also `findProviders`

131.6.3.3 public Map<Resource,Wiring> getWirings ( )

☐ An immutable map of wirings for resources. Multiple calls to this method for the same environment object must result in the same set of wirings.

Returns the wirings already defined in this environment

131.6.3.4 public boolean isEffective (Requirement requirement)

requirement the Requirement to test

☐ Test if a given requirement should be wired in a given resolve operation. If this method returns false then the resolver should ignore this requirement during this resolve operation.

The primary use case for this is to test the `effective` directive on the requirement, though implementations are free to use this for any other purposes.

Returns true if the requirement should be considered as part of this resolve operation

Throws `NullPointerException` – if requirement is null

131.6.4 public class ResolutionException extends RuntimeException

Indicates failure to resolve a set of requirements.

If a resolution failure is caused by a missing mandatory dependency a resolver may include any requirements it has considered in the resolution exception. Clients may access this set of dependencies via the `getUnresolvedRequirements()` method.

Resolver implementations may subclass this class to provide extra state information about the reason for the resolution failure.

Concurrency Thread-safe Immutable

131.6.4.1 public ResolutionException (String message , Throwable cause , Collection<Requirement> unresolvedRequirements)

message The message.

cause The cause of this exception.

unresolvedRequirements the requirements that are unresolved or null if no unresolved requirements information is provided.

☐ Creates an exception of type `ResolutionException`.

This method creates an `ResolutionException` object with the specified message, cause and unresolvedRequirements.

131.6.4.2 public ResolutionException (String message)

message The message.

□ Creates an exception of type ResolutionException.

This method creates an ResolutionException object with the specified message.

131.6.4.3 public ResolutionException (Throwable cause)

cause The cause of this exception.

□ Creates an exception of type ResolutionException.

This method creates an ResolutionException object with the specified cause.

131.6.4.4 public Collection<Requirement> getUnresolvedRequirements ( )

□ May contain one or more unresolved mandatory requirements from mandatory resources.

This exception is provided for informational purposes and the specific set of requirements that are returned after a resolve failure is not defined.

Returns a collection of requirements that are unsatisfied

131.6.5 public interface Resolver

A resolver is a service interface that can be used to find resolutions for specified resources based on a supplied Environment.

Concurrency Thread-safe

131.6.5.1 public Map<Resource,List<Wire>> resolve (Environment environment , Collection<? extends Resource> mandatoryResources , Collection<? extends Resource> optionalResources) throws ResolutionException

environment the environment into which to resolve the requirements

*mandatoryResources*The resources that must be resolved during this resolution step or null if no resources must be resolved

optionalResources Any resources which the resolver should attempt to resolve but that will not cause an exception if resolution is impossible or null if no resources are optional.

□ Attempt to resolve the resources based on the specified environment and return any new resources and wires to the caller.

The resolver considers two groups of resources:

- Mandatory - any resource in the mandatory group must be resolved, a failure to satisfy any mandatory requirement for these resources will result in a ResolutionException
- Optional - any resource in the optional group may be resolved, a failure to satisfy a mandatory requirement for a resource in this group will not fail the overall resolution but no resources or wires will be returned for this resource.

Delta

The resolve method returns the delta between the start state defined by Environment.getWirings() and the end resolved state, i.e. only new resources and wires are included. To get the complete resolution the caller can merge the start state and the delta using something like the following:

```
Map<Resource, List<Wire>> delta = resolver.resolve(env, resources, null);
Map<Resource, List<Wire>> wiring = env.getWiring();

for (Map.Entry<Resource, List<Wire>> e : delta.entrySet()) {
Resource res = e.getKey();
List<Wire> newWires = e.getValue();
List<Wire> currentWires = wiring.get(res);
if (currentWires != null) {
newWires.addAll(currentWires);
}

wiring.put(res, newWires);
}
```

Consistency

For a given resolve operation the parameters to the resolve method should be considered immutable. This means that resources should have constant capabilities and requirements and an environment should return a consistent set of capabilities, wires and effective requirements.

The behavior of the resolver is not defined if resources or the environment supply inconsistent information.

Returns the new resources and wires required to satisfy the requirements

Throws `ResolutionException` – if the resolution cannot be satisfied for any reason

`NullPointerException` – if environment is null

131.6.6 public interface Synthesized

During the resolution process it is necessary for the resolver to synthesize resources to represent the merge of bundles and fragments.

For example if we have one bundle A and two fragments X and Y then resolver will create two synthesized resources AX and AY.

In order for the Environment to provide a policy for which synthesized resources are preferred, both the host bundle and providing fragment need to be visible. This interface must therefore be used on all synthesized resource elements created by a resolver.

The environment can then enforce the ordering policy when a capability that is provided by a synthesized resource is added to the SortedSet returned by `Environment.findProviders(org.osgi.framework.resource.Requirement)`.

The following is a short example of how this may look in practice:

```
class PreferrSmallResourcesEnvironment implements Environment {
static class SizeComparator extends Comparator {
public int compare(Capability capA, Capability capB) {
Resource hostA = capA.getResource();
Resource hostB = capB.getResource();

Resource synthA = getSynthesizedResource(hostA);
Resource synthB = getSynthesizedResource(hostA);

long sizeA = getSize(hostA) + getSize(synthA);
long sizeB = getSize(hostB) + getSize(synthB);

if (sizeA > sizeB) {
return +1;
}
else
if (sizeA < sizeB) {
return -1;
}
else {
return 0;
}
}

private int getSize(Resource resource) {
if (resource == null)
return 0;

Capability contentCapability = getCapability(resource,
ContentNamespace.CAPABILITY);

if (contentCapability == null)
return 0;

Integer size = (Integer) contentCapability.getAttributes().get(
ContentNamespace.SIZE_ATTRIBUTE);

return size == null ? Integer.MAX_VALUE : size;
}

private Capability getCapability(Resource res, String namespace) {
List<Capability> caps = res.getCapabilities(namespace);
return caps.isEmpty() ? null : caps.get(0);
}

private Resource getSynthesizedResource(Resource res) {
if (res instanceof Synthesized) {
return (Resource) ((Synthesized) res).getOriginal();
}
else {
return null;
}
}
};

private List repositories;

public PreferrSmallResourcesEnvironment(List repositories) {
this.repositories = repositories;
}

public SortedSet findProviders(Requirement requirement) {
SortedSet caps = new ConcurrentSkipListSet(new SizeComparator());

for (Repository r : repositories) {
caps.addAll(r.findProviders(requirement));
}

return caps;
}

public boolean isEffective(Requirement requirement) {
return true;
}
```

```
public Map getWirings() {
return Collections.EMPTY_MAP;
}
}
```

A Synthesized capability must report the Host Bundle via the Capability.getResource() method. It must also report the original Capability from the Fragment Bundle via the getOriginal() method. The Environment may then access the fragment resource that this capability originates from via Capability.getResource().

131.6.6.1 public Object getOriginal ( )

□ The original Capability, Requirement or Resource that backs this synthesized element.

Returns the original capability, requirement or resource

131.7 References

[1] *Core Specification 4.3*
http://www.osgi.org/Download/Release4V43

[2] *Framework Wiring API (Core)*
http://www.osgi.org/Download/Release4V43

[3] *JSR 330: Dependency Injection Framework*
http://jcp.org/en/jsr/detail?id=330

[4] *Google Guice*
http://code.google.com/p/google-guice/

[5] *File Install*
http://felix.apache.org/site/apache-felix-file-install.html