Jakarta HiveMind Project Documentation

1. HiveMind Project

1.1. Reference

1.1.1. HiveMind Services

In HiveMind, a service is simply an object that implements a particular interface, the *service interface*. You supply the service interface (packaged as part of a module). You supply the core implementation of the interface (in the same module, or in a different module). At runtime, HiveMind puts it all together.

HiveMind uses four service models: primitive, singleton, threaded and pooled. In the primitive and singleton models, each service will ultimately be just a single object instance. In the threaded and pooled models, there may be many instances simultaneously, one for each thread.

Unlike EJBs, there's no concept of location transparency: services are always local to the same JVM. Unlike XML-based web services, there's no concept of language transparency: services are always expressed in terms of Java interfaces. Unlike JMX or Jini, there's no concept of hot-loading of of services. HiveMind is kept deliberately simple, yet still very powerful, so that your code is kept simple.

1.1.1.1. Defining Services

A service definition begins with a Java interface, the service interface. Any interface will do, HiveMind doesn't care, and there's no base HiveMind interface.

A module descriptor may include <service-point> elements to define services. A module may contain any number of services.

Each <service-point> establishes an id for the service and defines the interface for the service. An example is provided later in this document.

HiveMind is responsible for supplying the service implementation as needed; in most cases, the service implementation is an additional Java class which implements the service interface. HiveMind will instantiate the class and configure it as needed. The exact timing is determined from the service's service model:

- **primitive** : the service is constructed on first reference
- **singleton** : the service is not constructed until a method of the service interface is invoked
- **threaded** : invoking a service method constructs and binds an instance of the service to the current thread
- **pooled** : as with threaded, but service implementations are stored in a pool when unbound from a thread for future use in other threads.

Additional service models can be defined via the hivemind.ServiceModels configuration point.

HiveMind uses a system of *proxies* for most of the service models (all except the primitive service model, which primarily exists to bootstrap the core HiveMind services used by other services). Proxies are objects that implement the service interface and take care of details such as constructing the actual implementation of a service on the fly. These lifecycle issues are kept hidden from your code behind the proxies.

A service definition may include *service contributions*, or may leave that for another module.

Ultimately, a service will consist of a core implementation (a Java object that implements the service interface) and, optionally, any number of interceptors. Interceptors sit between the core implementation and the client, and add functionality to the core implementation such as logging, security, transaction demarkation or performance monitoring. Interceptors are yet more objects that implement the service interface.

Instantiating the core service implementation, configuring it, and wrapping it with any interceptors is referred to as *constructing the service*. Typically, a service proxy will be created first. The first time that a service method is invoked on the proxy, the service implementation is instantiated and configured, and any interceptors for the service are created.

1.1.1.2. Extending Services

Any module may contribute to any service extension point. An <implementation> element contains these contributions. Contributions take three forms:

- Service constructors:
  - <create-instance> to instantiate an instance of a Java class as the implementation
  - <invoke-factory> to have another service create the implementation
- <interceptor> to add additional logic to a core implementation

Service Constructors

A service constructor is used to instantiate a Java class as the core implementation instance for the service.

There are two forms of service constructors: instance creators and implementation factories.

An instance creator is represented by a <create-instance> element. It includes a class attribute, the Java class to instantiate.

An implementation factory is represented by a <invoke-factory> element. It includes a service-id attribute, the id of a service implementation factory service (which implements the ServiceImplementationFactory interface). The most common example is the hivemind.BuilderFactory service.

Implementation Factories

An implementation factory is used to create a core implementation for a service at runtime.

Often, the factory will need some additional configuration information. For example, the hivemind.lib.EJBProxyFactory service uses its parameters to identify the JNDI name of the EJB's home interface, as well as the home interface class itself.

Parameters to factory services are the XML elements enclosed by the <invoke-factory> element. Much like a configuration contribution, these parameters are converted from XML into Java objects before being provided to the factory.

The most common service factory is hivemind.BuilderFactory. It is used to construct a service and then set properties of the service implementation object.

Interceptor Contributions

An interceptor contribution is represented by an <interceptor> element. The service-id attribute identifies a service interceptor factory service: a service that implements the ServiceInterceptorFactory interface.

An interceptor factory knows how to create an object that implements an arbitrary interface (the interface being defined by the service extension point), adding new functionality. For example, the hivemind.LoggingInterceptor factory creates an instance that logs entry and exit to each method.

The factory shouldn't care what the service interface itself is ... it should adapt to whatever interface is defined by the service extension point it will create an instance for.

A service extension point may have any number of interceptor contributions. If the order in which interceptors are applied is important, then the optional `before` and `after` attributes can be specified.

A Stack of Interceptors

In this example, is was desired that any method logging occur first, before the other interceptors. This ensures that the time taken to log method entry and exit is not included in the performance statistics (gathered by the performance interceptor). To ensure that the logging interceptor is the first, or earliest, interceptor, the special value `*` (rather than a list of interceptor service ids) is given for its `before` attribute (within the <interceptor> element). This forces the logging interceptor to the front of the list (however, only a single interceptor may be so designated).

Likewise, the security checks should occur last, after logging and after performance; this is accomplished by setting the `after` attribute to `*`. The performance interceptor naturally falls between the two.

This is about as complex as an interceptor stack is likely to grow. However, through the use of explicit dependencies, almost any arraingment of interceptors is possible ... even when different modules contribute the interceptors.

Interceptors implement the `toString()` method to provide a useful identification for the interceptor, for example:
`< Interceptor: hivemind.LoggingInterceptor for com.myco.MyService(com.myco.MyServiceInterface)>`

This string identifies the interceptor service factory (hivemind.LoggingInterceptor), the service extension point (com.myco.MyService) and the service interface (com.myco.MyServiceInterface).

If `toString()` is part of the service interface (really, a very rare case), then the interceptor *does not* override the service implementation's method.

1.1.1.3. A short example

As an example, let's create an interface with a single method, used to add together two numbers.

```
package com.myco.mypackage;

public interface Adder
{
  public int add(int arg1, int arg2);
}
```

We could define many methods, and the methods could throw exceptions. Once more, HiveMind doesn't care.

We need to create a module to contain this service. We'll create a simple HiveMind deployment descriptor. This is an SDL file, named hivemodule.sdl, that must be included in the module's META-INF directory.

```
module (id=com.myco.mypackage version="1.0.0")
{
  service-point (id=Adder interface=com.myco.mypackage.Adder)
}
```

The complete id for this service is `com.myco.mypackage.Adder` , formed from the module id and the service id. Commonly, the service id will exactly match the complete name of the service interface, but this is not required.

Normally, the <service-point> would contain a <create-instance> or <invoke-factory> element, used to create the core implementation. For this example, we'll create a second module that provides the implementation. First we'll define the implementation class.

```
package com.myco.mypackage.impl;

import com.myco.mypackage.Adder;

public class AdderImpl implements Adder
{
  public int add(int arg1, int arg2)
  {
    return arg1 + arg2;
  }
}
```

That's what we meant by a POJO. We'll create a second module to provide this implementation.

```
module (id=com.myco.mypackage.impl version="1.0.0")
{
  implementation (service-id=com.myco.mypackage.Adder)
  {
    create-instance (class=com.myco.mypackage.impl.AdderImpl)
  }
}
```

The runtime code to access the service is very streamlined:

```
Registry registry = . . .
Adder service = (Adder) registry.getService("com.myco.mypackage.Adder", Adder.class);
int sum = service.add(4, 7);
```

Another module may provide an interceptor:

```
module (id=com.myco.anotherpackage version="1.0.0")
{
  implementation (service-id=com.myco.mypackage.Adder)
  {
    interceptor (service-id=hivemind.LoggingInterceptor)
  }
}
```

Here the Logging interceptor is applied to the service extension point. The interceptor will be inserted between the client code and the core implementation. The client in the code example won't get an instance of the AdderImpl class, it will get an instance of the interceptor, which internally invokes methods on the AdderImpl instance. Because we code against interfaces instead of implementations, the client code neither knows nor cares about this.

1.1.1.4. Primitive Service Model

The simplest service model is the **primitive** service model; in this model the service is constructed on first reference. This is appropriate for services such as service factories and interceptor factories, and for several of the basic services provided in the hivemind module.

1.1.1.5. Singleton Service Model

Constructing a service can be somewhat expensive; it involves instantiating a core service implementation, configuring its properties (some of which may also be services), and building the stack of interceptors for the service. Although HiveMind encourages you to define your application in terms of a large number of small, simple, testable services, it is also desirable to avoid a cascade of unneccesary object creation due to the dependencies between services.

To resolve this, HiveMind defers the actual creation of services by default. This is controled by the `model` attribute of the <service-point> element; the default model is **singleton**.

When a service is first requested a *proxy* for the service is created. This proxy implements the same service interface as the actual service and, the first time a method of the service interface is invoked, will force the construction of the actual service (with the core service implementation, interceptors, references to other services, and so forth).

In certain cases (including many of the fundamental services provided by HiveMind) this behavior is not desired; in those cases, the **primitive** service model is specified. In addition, there is rarely a need to defer service implementation or service interceptor factory services.

1.1.1.6. Threaded Service Model

In general, singleton services (using the singleton or primitive service models) should be sufficient. In some cases, the service may need to keep some specific state. State and multithreading don't mix, so the **threaded** service model constructs, as needed, a service instance for the current thread. Once constructed, the service instance stays bound to the thread until it is discarded. The particular service implementation is exclusive to the thread and is only accessible from that thread.

The threaded service model uses a special proxy class (fabricated at runtime) to support this behavior; the proxy may be shared between threads but methods invoked on the proxy are redirected to the private service implementation bound to the thread. Binding of a service implementation to a thread occurs automatically, the first time a service method is invoked.

The service instance is discarded when notified to cleanup; this is controlled by the hivemind.ThreadEventNotifier service. If your application has any threaded services, you are responsible for invoking the `fireThreadCleanup()` method of the service.

A core implementation may implement the Discardable interface. If so, it will receive a notification as the service instance is discarded.

HiveMind includes a servlet filter to take care creating the Registry and managing the ThreadEventNotifier service.

1.1.1.7. Pooled Service Model

The pooled service model is very similar to the threaded model, in that a service implementation will be exclusively bound to a particular thread (until the thread is cleaned up). Unlike the threaded model, the service is not discarded; instead it is stored into a pool for later reuse with the same or a different thread.

As with the threaded model, all of this binding and unbinding is hidden behind a dynamically fabricated proxy class.

Core service implementations may implement the RegistryShutdownListener interface to receive a callback for final cleanups (as with the singleton and deferred service models).

In addition, a service may implement the PoolManageable interface to receive callbacks specific to the pooled service. The service is notified when it is activated (bound to a thread) and deactivated (unbound from the thread and returned to the pool).

1.1.1.8. Service Lifecycle

As discussed, the service model determines when a service is instantiated. In many cases, the service needs to know when it has been created (to perform any final initializations) or when the Registry has been shut down.

A core service implementation may also implement the RegistryShutdownListener interface. When a Registry is shutdown, the `registryDidShutdown()` method is invoked on all services (and many other objects, such as proxies). The order in which these notifications occur is not defined. A service may release any resources it may hold at this time. It should not invoke methods on other service interfaces.

The threaded service model does **not** register services for Registry shutdown notification; regardless of whether the core service implementation implements the RegistryShutdownListener interface or not. Instead, the core service implementation should implement the Discardable interface, to be informed when a service bound to a thread is discarded.

It is preferred that, whenever possible, services use the singleton service model (the default) and not the primitive model. All the service models (except for the primitive service model) expose a **proxy** object (implementing the service interface) to client code (included other services). These proxies are aware of when the Registry is shutdown and will throw an exception when a service method is invoked on them.

1.1.1.9. Services and Events

It is fairly common that some services will produce events and other services will consume events. The use of the hivemind.BuilderFactory to construct a service simplifies this, using the `< event-listener>` element. The BuilderFactory can register a *core service implementation* (not the service itself!) as a *listener* of events produced by some other service.

The producing service must include a matched pair of listener registration methods, i.e., both `addFooListener()` and `removeFooListener`. Note that only the *implementation class* must implement the listener interface; the service interface does not have to extend the listener interface. The core service implementation is registered directly with the producer service, bypassing any interceptors or proxies.

1.1.1.10. Frequently Asked Questions

- **Why do I pass the interface class to getService()**?

  This is to add an additional level of error checking and reporting. HiveMind knows, from the module descriptors, the interface provided by the service extension point, but it can't tell if *you* know that. By passing in the interface you'll cast the returned service to, HiveMind can verify that you won't get a ClassCastException. Instead, it throws an exception with more details (the service extension point id, the actual interface provided, and the interface you passed it).

- **What if no module provides a core implementation of the service?**

  HiveMind checks for a service constructor when the registry itself is assembled. If a service extension point has no service constructor, an error is logged (identifying the extension point id). In addition, `getService()` will throw an ApplicationRuntimeException.
- **What if I need to do some initializations in my service?**

  If you have additional initializations that can't occur inside your core service implementations constructor (for instance, if the initializations are based on properties set after the service implementation object is instantiated), then your class should use the hivemind.BuilderFactory to invoke an initializer method.
- **What if I don't invoke Registry.cleanupThread()?**

  Then service implementations bound to the current thread stay bound. When the thread is next used to process a request, the same services, in whatever state they were left in, will be used. This may not be desirable in a servlet or Tapestry application, as some state from a client may be left inside the services, and a different client may be associated with the thread in later executions.
- **What if I want my service to be created early, not just when needed?**

  Contribute your service into the hivemind.EagerLoad configuration; this will force HiveMind to instantiate the service on startup. This is often used when developing an application, so that configuration errors are caught early; it may also be useful when a service should be instantiated to listen for events from some other service.

1.1.2. Configuration Points

A central concept in HiveMind is *configuration extension points*. Once you have a set of services, its natural to want to configure those services. In HiveMind, a configuration point contains an unordered list of *elements*. Each element is contributed by a module ... any module may make contributions to any configuration point.

There is no explicit connection between a service and a configuration point, though it is often the case that a service and a configuration point will be similarily named (or even identically named; services and configuration points are in seperate namespaces). Any relationship between a service and an configuration point is explicit only in code ... the service may be configured with the elements of a configuration point and operate on those elements in some way.

1.1.2.1. Defining a Configuration Point

A module may include <configuration-point> elements to define new configuration points. A configuration point may specify the expected, or allowed, number of contributions:

- Zero or one
- Zero or more (the default)
- At least one
- Exactly one

At runtime, the number of actual contributions is checked against the constraint and an error is reported if the number doesn't match.

Defining the Contribution Format

A significant portion of an configuration point is the <schema> element ... this is used to define the format of contributions that may be made inside <contribution> elements. Contributions take the form of XML elements and attributes, the <schema> element identifies which elements and which attributes and provides rules that transform the contributions into Java objects.

This is very important: what gets fed into an configuration point (in the form of contributed <contribution>s) is XML. What comes out on the other side is a list of configured Java objects. Without these XML transformation rules, it would be necessary to write Java code to walk the tree of XML elements and attributes to create the Java objects; instead this is done inside the module deployment descriptor, by specifying a <schema> for the configuration point, and providing rules for processing each contributed element.

If a contribution from an <contribution> is invalid, then a runtime error is logged and the contribution is ignored. The runtime error will identify the exact location (the file, line number and column number) of the contribution so you can go fix it.

The <schema> element contains <element> elements to describe the XML elements that may be contributed. <element>s contain <attribute>s to define the attributes allowed for those elements. <element>s also contain <rules> used to convert the contributed XML into Java objects.

Here's an example from the HiveMind test suite. The `Datum` object has two properties: key and value.

```
configuration-point (id=Simple)
{
  schema
  {
    element (name=datum)
    {
      attribute (name=key required=true)
      attribute (name=value required=true)
      conversion (class=hivemind.test.config.impl.Datum)
    }
  }
}

contribution (configuration-id=Simple)
{
  datum (key=key1 value=value1)
  datum (key="another key" value=<<A value with a "quote" in it>>)
}
```

The <conversion> element creates an instance of the class, and initializes its properties from the attributes of the contributed element (the `datum` and its `key` and `value` attributes). For more complex data, the <map> and <rules> elements add power (and complexity).

This extra work in the module descriptor eliminates a large amount of custom Java code that would otherwise be necessary to walk the XML contributions tree and convert elements and attributes into objects and properties. Yes, you could do this in your own code ... but would you really include all the error checking that HiveMind does? Or the line-precise error reporting? Would you bother to create unit tests for all the failure conditions?

Using HiveMind allows you to write the schema and rules and know that the conversion from XML to Java objects is done uniformly, efficiently and robustly.

The end result of this mechanism is very concise, readable contributions (as shown in the <contribution> in the example).

In addition, it is common for multiple configuration points to share the exact same schema. By assigning an id attribute to a <schema> element, you may reference the same schema for multiple configuration points. For example, the hivemind.FactoryDefaults and hivemind.ApplicationDefaults configuration points use the same schema. The hivemind module deployment descriptor accomplishes this by defining a schema for one configuration point, then referencing it from another:

```

schema (id=Defaults)
{
  element (name=default)
  {
    . . .
  }
}

configuration-point (id=FactoryDefaults schema-id=Defaults)
```

Like service points and configuration points, schemas may be referenced within a single module using an unqualified id, or referenced between modules using a fully qualified id (that is, prefixed with the module's id).

1.1.2.2. Accessing Configuration Points

Like services, configuration points are meant to be easy to access (the only trick is getting a reference to the registry to start from).

```
Registry registry = . . .;
List elements = registry.getConfiguration("com.myco.MyConfig");

int count = elements.size();
for (int i = 0; i < count; i++)
{
  MyElement element = (MyElement) elements.get(i);

  . . .
}
```

Note:

Although it is possible to access configurations via the Registry, it is often not a good idea. It is unlikely that you want the information contained in a configuration as an unordered list. A best practice is to always access the configuration through a service, which can organize and validate the data in the configuration.

The list of elements is always returned as an unmodifiable list. An empty list may be returned.

The order of the elements in the list is not defined. If order is important, you should create a new (modifiable) list from the returned list and sort it.

Note that the elements in the list are no longer the XML elements and attributes that were contributed, the rules provided in the configuration point's <schema> are used to convert the contributed XML into Java objects.

1.1.2.3. Lazy Loading

At application startup, all the module deployment descriptors are located and parsed and in-memory objects created. Validations (such as having the correct number of contributions) occur at this stage.

The list of elements for an configuration point is not created until the first call to `Registry.getConfiguration()` for that configuration point.

In fact, it is not created even then. When the element list for an configuration point is first accessed, what's returned is not *really* the list of elements; it's a proxy, a stand-in for the real data. The actual elements are not converted until they are actually needed, in much the same way that the creation of services is deferred.

In general, you will never know (or need to know) this; when you access the `size()` of the list or `get()` any of its elements, the conversion of contributions into Java objects will be triggered, and those Java objects will be returned in the list.

If there are minor errors in the contribution, then you may see errors logged; if the <contribution> contributions are singificantly malformed, HiveMind may be unable to recover and will throw a runtime exception.

1.1.2.4. Substitution Symbols

The information provided by HiveMind module descriptors is entirely static, but in some cases, some aspects of the configuration should be dynamic. For example, a database URL or an e-mail address may not be known until runtime (a sophisticated application may have an installer which collects this information).

HiveMind supports this notion through *substitution symbols*. These are references to values that are supplied at runtime. Substitution symbols can appear inside literal values ... both as XML attributes, and as character data inside XML elements.

Example:

```
contribution (configuration-id=com.myco.MyConfig)
{
  value { "dir/foo.txt" }
  value { "${config.dir}/${config.file}" }
}
```

This example contributes two elements to the `com.myco.MyConfig` configuration point. The first contribution is simply the text `dir/foo.txt`. In the second contribution, the content contains substitution symbols (which use a syntax derived from the Ant build tool). Symbol substitution occurs *before* <schema> rules are executed, so the `config.dir` and `config.file` symbols will be converted to strings first, then whatever rules are in place to convert the `value` element into a Java object will be executed.

Symbol Sources

This begs the question: where do symbol values come from? The answswer is application dependent. HiveMind itself defines a configuration configuration point for this purpose: hivemind.SymbolSources. Contributions to this configuration point define new objects that can provide values for symbols, and identify the order in which these objects should be consulted.

If at runtime none of the configured SymbolSources provides a value for a given symbol then HiveMind will leave the reference to that symbol *as is*, including the surrounding `${` and `}`.

Additionally an error will be logged.

1.1.2.5. Frequently Asked Questions

- **Are the any default implementations of SymbolSource?**

  There is now an configuration point for setting factory defaults: hivemind.FactoryDefaults . A second configuration point, for application defaults, overrides the factory defaults: hivemind.ApplicationDefaults.

  SystemPropertiesSymbolSource is a one-line implementation that allows access to system properties as substitution symbols. Note that this configuration is *not* loaded by default.

  Additional implementations may follow in the future.

- **What's all this about schemas and rules?**

  A central goal of HiveMind is to reduce code clutter. If configuration point contributions are just strings (in a .properties file) or just XML, that puts a lot of burden on the developer whose code *reads* the configuration to then massage it into useful objects. That kind of ad-hoc code is notoriously buggy; in HiveMind it is almost entirely absent. Instead, all the XML parsing occurs inside HiveMind, which uses the schema and rules to validate and convert the XML contributions into Java objects.

  You can omit the schema, in which case the elements are left as XML (instances of Element and your code is responsible for walking the elements and attributes ... but why bother? Far easier to let HiveMind do the conversions and validations.

- **How do I know if the element list is a proxy or not?**

  Basically, you can't, short of performing an `instanceof` check. There isn't any need to tell the difference between the deferred proxy to the element list and the actual element list; they are both immutable and both behave identically.

1.1.3. Simple Data Language

One of the frequent criticisms of J2EE is: *too much XML*. That is, every small aspect of a J2EE application requires a big XML deployment descriptor to be generated (by hand, or generated from code in some way). What's interesting is what's *in* those XML files: configuration data containing simple strings and identifiers.

XML is overkill for these purposes: its a *markup* language, designed to add *semantic meaning* to documents that normally have a *literal* meaning (that is, documents that are supposed to be read primarily by persons, not other programs). Like many technologies, its intended use has been co-opted (to what degree is debatable). XML for real documents such as XHTML or SVG make sense. The complexity of SOAP mandates an industrial strength syntax to express its complex structure. But for the majority of uses of XML within the J2EE stack, it simply is vastly more complex than is necessary.

The complexity comes at some cost ... XML is very verbose, a tangle of punctuation (such as **<**, **>** and quotes) and repetition (start tags and end tags). Even experienced developers often need to take a bit of time to visually and mentally parse an XML snippet.

Through HiveMind release 1.0-alpha-4, HiveMind was as guilty as the next project in XML usage. HiveMind module deployment descriptors would, at least, centralize the XML concerning a service, and enforce some amount of uniformity.

Release 1.0-alpha-5 introduces **Simple Data Language**, an *alternative* to the use of XML in HiveMind. XML will continue to be supported as a first class citizen, but in HiveMind, there is not such a compelling reason to use it!

1.1.3.1. Goals

The goals of SDL are to provide the bare essentials needed for a hierachical data language, but keep is spare and readable. Unecessary typing is to be avoided, so the use of quotes is made optional whereever possible. SDL syntax should be reasonably obvious to an interested observer.

1.1.3.2. Examples

Before getting bogged down in a formal specification for SDL, a few simple examples will explain just about everything. Compare the following two HiveMind module deployment descriptors, which express identical information:

Traditional XML Format:

```
<?xml version="1.0"?>

<module id="some.module" version="1.0.0">
  <configuration id="ControlPipeline">
    <schema>
      <element name="processor">

         <attribute name="name" required="true"/>
         <attribute name="service-id" required="true" translator="service"/>
         <attribute name="before"/>
         <attribute name="after"/>

         <conversion class="some.module.PipelineContribution">
           <map property="controlService" attribute="service-id"/>
         </conversion>
      </element>
    </schema>
  </configuration>
</module>
```

SDL format:

```
module (id=some.module version="1.0.0")
{
  configuration (id=ControlPipeline)
  {
    schema
    {
      element (name=processor)
      {
        attribute (name=name required=true)
        attribute (name=service-id required=true translator=service)
        attribute (name=before)
        attribute (name=after)

        conversion (class=some.module.PipelineContribution)
        {
          map (property=controlService attribute=service-id)
        }
      }
    }
  }
}
```

Some observations:

- SDL uses open and close braces to denote containment of elements within another element
- Attributes, as a list of name-value pairs, are placed in parenthesis following the element name
- Elements without attributes can omit the parenthesis (example: `schema`)
- Elements that do not contain other elements can omit the open and close braces denoting thier body (example: `attribute`)
- Most common strings do not have to be quoted
- *All* whitespace not inside quotes is ignored

1.1.3.3. Whitespace

All whitespace (outside of literals) is ingored. Whitespace is considered to be:

- Spaces
- Tabs
- Newlines

- Carriage Returns

1.1.3.4. Comments

Comments are in the format traditional in Java and C:

```
// This is a comment that extends to the end of the current line.

/* This is a multiline
   comment. */
```

Comments may appear anywhere in an SDL document (except within quoted strings) and are always ignored.

1.1.3.5. Element and Attribute Names

Element and attribute names must be *simple ids*. They must start with a letter (or underscore) and may contain only letters, digits, underscores and dashes. They may *not* be enclosed in quotes.

1.1.3.6. Literal Values

Attribute values may be literal values. Literal values are considered one of the following:

- simple ids
- complex ids -- a sequence of simple ids seperated by periods
- segmented ids -- a sequence of complex ids separated by colons
- numeric values
- Symbol references
- Quoted strings
- Extended literals

Complex ids have the same format as Java class and interface names (but can, additionally, contain dash characters which are not allowed in Java).

Numeric values consist of an optional sign (`+` or `-`) followed by a integer or decimal value. In the future, a more expansive definition may be provided.

Symbol references allow Ant-style symbols to be used directly in SDL. Example:

```
. . .
  set-service (service-id=${symbol.for.service-id})
. . .
```

Support for Ant-style symbols is a convienience (the same syntax is used heavily within HiveMind). There is no difference between `${symbol.for.service-id}` and

`"${symbol.for.service-id}"` ... both will be processed identically.

Quoted strings are similar to Java string literals. All whitespace within the string is retained as-is, including line breaks. A subset of the Java escape codes are currently supported:

- \t (tab)
- \n (newline)
- \r (carriage return)
- \" (quote)
- \\ (slash)

Any other sequence is passed through normally (unescaped).

Extended literals have a different syntax:

```
. . .
  description =
<< A long, multiline string
that may contain "quoted" sections. >>
. . .
```

Extended literals may contain any character sequence (except `>>`). Escape sequences in expanded literals are not interpreted. All whitespace within the delimiters is retained.

1.1.3.7. Literal Gotcha

The body of an element may contain literal text data, just as with XML. Unlike XML, whitespace is completely removed. Thus the following are equivalent:

```
first
{
  "NowIsTheTime"
}
second
{
  "Now" "Is" <<The>> <<Time>>
}
```

This applies to all forms of literals, including numbers. The following are identical:

```
pi1
{
  3.14159
}
pi2
{
  3 .14 159
}
```

Inside the body of an element, simple ids are interpreted as *elements* not string literals. In the following example, `root1` and `root2` have the same structure (each contains three children and no content). `leaf` contains no children, and its content is `child1child2child3`.

```
root1
{
  child1 {}
  child2 {}
  child3 {}
}
root2
{
  child1 child2 child3
}
leaf
{
  "child1" "child2" "child3"
}
```

1.1.3.8. TO DO

- Expand the definition of "character" to properly include Unicode
- Add Unicode escape patterns in quoted literals
- Expand the definition of numeric literal to include all Java literals

1.1.4. HiveDoc

HiveMind includes tools for documentating a HiveMind registry ... the combined information from all modules that are deployed at runtime. At build time, all related HiveMind module deployment descriptors are parsed and the results combined into a single file. The master file (which is only used for this documentation) is then converted into a set of HTML files using XSLT. The end result is much like JavaDoc ... it's fully hyperlinked and allows you to see all services, configuration points, contributions and schemas clearly.

Incorporated into the generated documentation is user-supplied descriptions. The <attribute>, <configuration-point>, <element>, <module>, <schema> and <service-point> elements can enclose a description (as character data), i.e.:

```
module (id=mymodule version="1.0.0")
{
  "A module for my application with my services, etc."
}
```

The HiveDoc for the HiveMind framework and library is available here.

Note:

Details on building the documentation will be coming soon.

1.1.5. HiveMind Module Descriptor

The purpose of the module descriptor is to provide a runtime and compile-time description of each HiveMind module in terms of service and configuration extension points and contributions to those extension points.

The descriptor is named `hivemodule.sdl` and is stored in the META-INF directory of the module.

The root element of the descriptor is the <module> element.

The prefered format is Simple Data Language, in which case, the descriptor should be named `hivemodule.sdl`. Alternately, you may name the file `hivemodule.xml` and use XML. *For the purposes of HiveMind,* the two formats are interchangeable.

1.1.5.1. attribute

<attribute> is used to define an attribute within an <element>. Inside a <contribution>, only known attributes are allowed in elements; unknown attributes will be logged as an error and ignored. In addition, some attributes are required; again, errors occur if the contributed element does not provide a value for the attribute.

Attribute	Type	Required ?	Description
name	string	yes	The name of the attribute.
required	boolean	boolean	If true, the attribute must be provided in the contributed configuration element. The default is false.
translator	string	no	The translator configuration that is used to convert the attribute into a useable type. By default, the attribute is treated as a single string.

1.1.5.2. configuration-point

The <configuration-point> element defines a configuration extension point.

Attribute	Type	Required ?	Description

<table>
<tr><td>id</td><td>string</td><td>yes</td><td>The simple id of the service extension point. The fully qualified id for the extension point is created by prefixing with the module's id (and a dot).</td></tr>
<tr><td>occurs</td><td><code>unbounded | 0..1 | 1 | 1..n</code></td><td>no</td><td>The number of contributions allowed:
- <b>unbounded</b> (default): any number
- <b>0..1</b>: optional
- <b>1</b> : required
- <b>1..n</b>: at least one</td></tr>
<tr><td>schema-id</td><td>string</td><td>no</td><td>Used to reference a <schema> (in the same module, or a different one) that defines the format of contributions into the configuration point. This may be omitted, in which case the extension point will contain a list of Element .</td></tr>
</table>

Contains: <schema>

<configuration-point> only defines a configuration point, it does not supply any data into that point. The <contribution> element is used to provide such data.

1.1.5.3. contribution

The <contribution> element contributes elements to an existing configuration extension point.

Attribute	Type	Required ?	Description
configuration-id	string	yes	Either the id of an <configuration-point> within the module, or the fully qualified id of an

			<configuration-point> in another module.

The content of the <contribution> consists of elements. These elements are converted, in accordance with the configuration point's <schema>, into Java objects.

1.1.5.4. conversion

<conversion> is an alternative to <rules> that is generally simpler and more concise. An <element> should contain <conversion> or <rules> but not both.

<conversion> is geared towards the typical case; a straight-forward mapping of the element to an instance of a Java class, and a mapping of the element's attributes to the properties of the Java object.

Attribute	Type	Required ?	Description
class	string	yes	The fully qualified name of a Java class to instantiate.
parent-method	string	no	The name of a method of the parent object used to add the created object to the parent. The default, `addElement`, is appropriate for top-level <element>s.

Contains: <map>

Each attribute will be mapped to a property. A limited amount of name mangling occurs: if the attribute name contains dashes, they are removed, and the character following is converted to upper case. So, an attribute named "complex-attribute-name" would be mapped to a property named "complexAttributeName". Only attributes identified with a <attribute> element will be mapped, others will be ignored.

1.1.5.5. create-instance

<create-instance> is used, within <service-point> and <implementation> to create the core service implementation for a service by instantiating a class. This is appropriate for simple services that require no explicit configuration.

Attribute	Type	Required ?	Description

class	class name	yes	Fully qualified class name to instantiate.
model	`primitive | singleton | threaded | pooled`	no	The model used to construct and manage the service. **singleton** is the default.

Additional service models can be defined via the hivemind.ServiceModels configuration point.

1.1.5.6. element

The <element> element is used to define an element in a the <schema>. <element> may also be nested within another <element>, to indicate an element that may be enclosed within another element.

Attribute	Type	Required ?	Description
name	string	yes	The name of the element.
content-translator	string	no	The translator configuration that is used to convert the element's content into a useable type. By default, the content is treated as a single string.

Contains: <attribute>, <conversion>, <element>, <rules>

Future enhancements to the HiveMind framework will include greater sophistication in defining configuration content.

1.1.5.7. implementation

The <implementation> element contributes a core implementation or interceptors to a service extension point.

Attribute	Type	Required ?	Description
service-id	string	yes	The id of the service to extend; this may be a fully qualified id, or the local id of a service

			within the same module.

Contains: <create-instance>, <interceptor>, <invoke-factory>

1.1.5.8. interceptor

<interceptor> contributes an interceptor factory to a service extension point. An interceptor factory is a service which implements the ServiceInterceptorFactory interface.

When the service is constructed, each invoked interceptor factory will fabricate an interceptor class to provide additional functionality for the service.

Attribute	Type	Required ?	Description
service-id	service id	yes	The id of the service that will create the interceptor for the service. This may be the local id of a service defined within the same module, or a fully qualified id.
before	string	no	A list of interceptors whose behavior should come later in execution than this interceptor.
after	string	no	A list of interceptors whose behavior should come earlier in execution than this interceptor.

Like a service implementation factory, a service interceptor factory may need parameters. As with <invoke-factory>, parameters to the interceptor factory are enclosed by the <interceptor> element. The service interceptor factory will decode the parameters using the schema identified by its `parameters-schema-id` attribute.

Interceptor ordering is based on dependencies; each interceptor can identify, by interceptor service id, other interceptors. Interceptors in the `before` list are deferred until after this interceptor. Likewise, this interceptor is deferred until after all interceptors in the `after` list.

Note:

The `after` dependencies will look familar to anyone who has used Ant or any version of Make. `before` dependencies are simply the opposite.

The value for `before` or `after` is a list of service ids seperated by commas. Service ids may be unqualified if they are within the same module. Alternately, the fixed value `*` may be used instead of a list: this indicates that the interceptor should be ordered absolutely first or absolutely last.

1.1.5.9. invoke-factory

The <invoke-factory> element is used to provide a service implementation for a service by invoking another service, a factory service.

Attribute	Type	Required ?	Description
service-id	string	yes	The id of the factory service. This may be a simple id for a service within the same module, or a fully qualified service id.
model	`primitive | singleton | threaded | pooled`	no	The model used to construct and manage the service. **singleton** is the default.

A service factory defines its parameters in terms of a schema. The content of the <invoke-factory> is converted, in accordance with the factory's schema, and provided to the factory.

Additional service models can be defined via the hivemind.ServiceModels configuration point.

1.1.5.10. map

The <map> element appears within <conversion> to override the default mapping from an attribute to a property. By default, the property name is expected to match the attribute name (with the name mangling described in the description of <conversion>); the <map> element is used to handle exceptions to the rule.

Attribute	Type	Required ?	Description
attribute	string	yes	The name of the attribute, which should match a name defined by an<attribute> (of the enclosing <element>).

property	string	yes	The corresponding property (of the Java object specified by the enclosing <conversion>)

1.1.5.11. module

The <module> element is the root element.

Attribute	Type	Required ?	Description
id	string	yes	The id should be a dotted sequence, like a package name. In general, the module id should *be* the package name.
version	version number	yes	The version of the module as a dotted sequence of three numbers. Example: "1.0.0"

Contains: <contribution>, <configuration-point>, <implementation> , <service-point>, <sub-module>

Note:
The version is not currently used, but a later release of HiveMind will include runtime dependency checking based on version number.

1.1.5.12. rules

<rules> is a container for element and attribute parsing rules within an <element>. These rules are responsible for converting the contributed element and its attributes into objects and object properties. The available rules are documented separately .

1.1.5.13. schema

The <schema> element is used to describe the format of element contributions to an <configuration-point>, or parameters provided to a service or interceptor factory.

Attribute	Type	Required ?	Description
id	string	yes	Assigns a local id to

			the schema that may be referenced elsewhere.

Contains: <element>

At a future time, the <schema> element will be extended to provide more options, to provide more precise control over the elements that may be provided in an <contribution>. At this time, a <schema> is simply a list of <element> elements.

> **Note:**
>
> When <schema> appears directly within <configuration-point>, or <parameters-schema> appears directly within <service-point>, then the `id` attribute is not allowed.

1.1.5.14. service-point

The <service-point> element defines a service extension point.

Attribute	Type	Required ?	Description
id	string	yes	The simple id of the service extension point. The fully qualified id for the extension point is created by prefixing with the module's id (and a dot).
interface	class name	yes	The fully qualified name of the Java interface supplied by this service extension point.
parameters-schema-id	string	no	Used to reference a <schema> (in the same module, or a different one) that defines parameters used by the service. This is used when the service being defined is a ServiceImplementationFactory or a ServiceInterceptorFactory.

Contains: <create-instance>, <interceptor>, <invoke-factory> , <parameters-schema>

1.1.5.15. sub-module

The <sub-module> element is used to identify an additional HiveMind module deployment descriptor. This is used when a single JAR file contains logically distinct packages, each of which should be treated as an individual HiveMind module. This can also be useful as a way to reduce developer conflict against a single, large, central module descriptor by effectively breaking it into smaller pieces. Sub-modules identified in this way must still have their own unique module id.

Attribute	Type	Required ?	Description
descriptor	string	yes	Location of the module descriptor.

The descriptor should be specified as a relative path, either the name of another module descriptor within the same folder, or within a child folder.

1.1.6. Contribution Processing Rules

The concept of performing a rules-directed conversion of elements and attributes into Java objects was pioneered (to my knowledge) in the Jakarta Digester framework (which started inside Tomcat, moved to Struts, and is now available on its own).

The technique is very powerful, even in the limited subset of Digester provided by HiveMind (over time, the number of available rules will increase).

1.1.6.1. Rules

Rules are attached to <element>s. Each rule object has two methods: the begin() method is invoked when the element is first encountered. The content of the element is then processed recursively (which will involve more rules). Once that completes, the end() method is invoked.

Note: `begin()` is invoked in the order that rules are defined within the <rules> element. `end()` is invoked in inverse order. This rarely makes any difference.

Element processing is based on an object stack. Several rules will manipulate the top object on the stack, setting properties based on attributes or content. The <create-object> rule will instantiate a new object at `begin()` and pop it off the stack at `end()` .

In several cases, rule descriptions reference the parent and child objects. The top object on the stack is the child, the object beneath that is the parent. The <set-parent> and

<invoke-parent> rules are useful for creating hierarchies of objects.

create-object

The <create-object> rule is used to create a new object, which is pushed onto the stack at `begin()`. The object is popped off the stack at `end()`. <create-object> is typically paired up with <invoke-parent> to connect the new object (as a child) to a parent object.

Attribute	Type	Required ?	Description
class	string	yes	The complete class name of the object to create. The class must be public, and have a no-arguments public constructor.

custom

The <custom> rule is used to provide a custom implementation of the `Rule` interface. Note that any such rules must not contain any individual state, as they will be reused, possibly by multiple threads.

Attribute	Type	Required ?	Description
class	string	yes	The complete class name of the class implementing the `Rule` interface.

invoke-parent

The <invoke-parent> rule is used to connect the child (top object on the stack) to its parent (the next object down). A method of the parent is invoked, passing the child as a parameter. This invocation occurs inside the rule's `begin()` method; to ensure that the child object is fully configured before being added to the parent place this rule after all properties of the child object have been configured.

Attribute	Type	Required ?	Description
method	string	yes	The name of the method to invoke on the parent object.
depth	number	no	The depth of the parent object within the object stack. The top

			object (the child) is at depth 0, and default depth of the parent is 1.

Note:

Top level elements should include an <invoke-parent> rule, and specify the method as `addElement`. This adds the created, configured object to the list of contributed objects for the <contribution> (or for service factories, adds the object as a parameter).

push-attribute

The <push-attribute> rule reads an attribute, converts it with a translator, and pushes the result onto the stack. It will typically be combined with a <invoke-parent> to get the pushed value added to the configuration point elements (or to some parent object).

Attribute	Type	Required ?	Description
attribute	string	yes	The name of the attribute to read.

read-attribute

The <read-attribute> rule reads an attribute from the current element, optionally translates it (from a string to some other type), and then assigns the value to a property of the top object on the object stack.

Attribute	Type	Required ?	Description
property	string	yes	The name of the property of the top object on the stack to update.
attribute	string	yes	The name of the attribute to read.
skip-if-null	boolean	no	If "true" (the default), then an omitted attribute will be ignored. If "false", the property will be updated regardless.
translator	string	no	A translator that overrides the attribute's translator.

read-content

The <read-content> rule is similar to <read-attribute>, except it concerns the content of the current element (the text wrapped by its start and end tags).

Attribute	Type	Required ?	Description
property	string	yes	The name of the property of the top object on the stack to update.

set-module

<set-module> is used to set a property of the top object on the stack to the module which made the contribution. This is often used when some other attribute of the contribution is the name of a service or configuration extension point (but it is advantageous to defer access to the service or configuration). The module can be used to resolve names of services or configurations that are local to the contributing module.

Attribute	Type	Required ?	Description
property	string	yes	The name of the property of the top object to update with the contributing module.

set-parent

The <set-parent> rule is used to set a property of the child object to parent object. This allows for backwards connections from child objects to parent objects.

Attribute	Type	Required ?	Description
property	string	yes	The name of the property of the child object to set.

set-property

The <set-property> rule is used to set a property of the top object to a preset value.

Attribute	Type	Required ?	Description
property	string	yes	The name of the

			property of the child object to set.
value	string	yes	The value to set the proeprty to. The is interpreted as with the smart translator, meaning that conversion to normal Java types (boolean, int, etc.) will work as expected.

1.1.6.2. Translators

Commonly, it is necessary to perform some translation or transformation of string attribute value to convert the value into some other type, such as boolean, integer or date. This can be accomplished by specifying a translator in the <attribute> element (it also applies to element content, with the `content-translator` attribute of the <element> element).

A translator is an object implementing the Translator interface. The `translator` value specified in a rule may be either the complete class name of a class implementing the interface, or one of a number of builtin values.

Translators configurations consist of a translator name, and an optional initalizer string. The initializer string is separated from the translator id by a comma, ex: `int,min=0` (where `min=0` is the initializer string). Initializer strings are generally in the format of `key=value[,key=value]*` ... but each Translator is free to interpret the initializer string its own way.

The following sections describe the basic translators provided with the framework. You can add additional translators by contributing to the hivemind.Translators configuration point.

bean

The bean translator expects its input to bean in the form `service-id:locator`. The service-id references a service implementing BeanFactory.

Note:
This translator is contributed by the hivemind.lib module.

boolean

The boolean translator converts an input string into a boolean value. "true" is translated to true, and "false" to false.

A default value is used when the input is blank. Normally, this default is false, but the "default" key in the initializer can override this (i.e., `boolean,default=true`).

class

The class translator converts a class name into a Class object. The value must be a fully qualified class name. A null input value returns null.

> **Note:**
>
> This translator is hard-coded, and does not appear in the hivemind.Translators configuration point.

configuration

The configuration translator converts an input value into a configuration point id, then obtains the elements for that configuration point as a List. The id may be fully qualified, or a local id within the contributing module.

A blank input value returns null.

double

The double translator converts the input into an double precision floating point value. It recognizes three initializer values:

- default: the default value (normally 0) to use when the input is blank
- min: a minimum acceptible value
- max: a maximum acceptible value

enumeration

The enumeration translator converts input strings into enumerated values. Enumeration *requires* an initializer string, with a special format:
`enumeration,class-name,input=field-name[,input=field-name]*`

That is, the initializer begins with the name of the class containing some number of public static fields. Input values are mapped against field names. Example:
`enumeration,java.lang.Boolean,yes=TRUE,no=FALSE`

If the input is null or empty, then the translator returns null.

id-list

Translates a comma-seperated list of ids into a comma-seperated list of fully qualified ids (qualified against the contributing module). Alternately, passes the value `*` through as-is. Id lists are typically used to establish ordering of items within a list, as with <interceptor>.

instance

The object translator converts a fully qualified class name into an object instance. The class must implement a public no-arguments constructor.

int

The int translator converts the input into an integer value. It recognizes three initializer values:

- default: the default value (normally 0) to use when the input is blank
- min: a minimum acceptable value
- max: a maximum acceptable value

> **Note:**
> This translator is hard-coded, and does not appear in the hivemind.Translators configuration point.

long

The long translator converts the input into an long integer (64 bit) value. It recognizes three initializer values:

- default: the default value (normally 0) to use when the input is blank
- min: a minimum acceptable value
- max: a maximum acceptable value

object

The object translator is allows the caller to provide an object value in a multitude of ways. The object translator inverts the normal roles; the caller has all the power in determining how to interpret the value, and the schema takes whatever value shows up. The object translator is driven by the hivemind.ObjectProviders configuration.

qualified-id

Translates an id into a fully qualified id (qualified against the contributing module's id).

resource

The resource translator is used to find a resource packaged with (or near) the module's deployment descriptor. The input value is the relative path to a file. The translator converts the input value to a Resource for that file.

If the file doesn't exist, then an error is logged. If a localization of the file exists, then the Resource for that localization is returned.

service

The service translator is used to lookup a service in the registry. The input value is either a local service id from the contributing module, or a fully qualified service id.

> **Note:**
> This translator is hard-coded, and does not appear in the hivemind.Translators configuration point.

service-point

The service translator is used to lookup a service point (not a service) in the registry. The input value is either a local service id from the contributing module, or a fully qualified service id.

smart

The smart translator attempts an automatic conversion from a string value (the attribute value or element content) to a particular type. It determines the type from the property to which the value will be assigned. Smart translator makes use of the JavaBeans's PropertyEditor class for the conversion, which allows easy this translator to be used with most common primitive types, such as int, short and boolean. See the SmartTranslator documentation for more details.

In general, the smart translator is the useful for most ordinary Java type properties, unless you want to specify range constraints.

It recognizes one initializer value:

- default: the default value to use when the input is blank

> **Note:**
> This translator is hard-coded, and does not appear in the hivemind.Translators configuration point.

1.1.7. Library Dependencies

HiveMind has a number of dependencies on other open-source frameworks. The Ant build files for HiveMind will automatically download dependencies from the Maven repository on ibiblio.

File	Name	Notes
commons-logging-1.0.3.jar	Commons-Logging	
easymock-1.1.jar	EasyMock testing framework	Only needed by HiveMindTestCase, which exists as the basis of your own tests.
jboss-j2ee-3.2.1.jar	JBoss J2EE Server	Used by some services of the HiveMind library. No dependencies on JBoss itself, just on the `javax.ejb` package.
javassist-2.6.jar	Javassist bytecode library	
oro-2.0.6.jar	ORO Regular Expressions	
spring-full-1.0.1.jar	Spring	Used by the hivemind.lib.SpringLookupFactory service.

In most cases, HiveMind has been built against a "handy" version; in most cases, you can vary the exact version of a dependency to suite your existing environment. Just remember to write some tests!

1.2. History of Changes

RSS

1.2.1. Version 1.0.beta-2 (unreleased)

- Removed dependency on Werkz. (KW) Fixes HIVEMIND-6.
- Added link to the Jakarta mailing lists page. (HLS)
- Modifed the build scripts to properly include variable info when compiling. (HLS) Thanks to Achim Hügen. Fixes HIVEMIND-21.
- Moved the Ant build scripts to a new directory, hivebuild, in preparation for making hivebuild reusable on new projects. (HLS)

- Added protected method constructRegistry() to HiveMindFilter. (HLS)
- Renamed existing 'object' translator to 'instance', and created a new 'object' translator with great flexibility. Extend BuilderFactory to add a set-object element that leverages the object translator. (HLS)
- Created service-property object translator. (HLS)
- Added a version of `Registry.getService()` that omits the service id (but requires that exactly one service point implements the service interface). (HLS) Thanks to Marcus Brito. Fixes HIVEMIND-20.
- Extended the BuilderFactory to autowire services. (HLS) Fixes HIVEMIND-22.
- Added a new module that contains HiveMind example code. (HLS)
- Fixed some latent bugs related to submodules inside the constructRegistry task. Made some more improvements to the hivebuild scripts. (HLS)
- Updated the download location for the Forrest distribution. (HLS)
- Added more examples and examples documentation. (HLS)
- Added StrictErrorHandler, an implementation of ErrorHandler that always throws an ApplicationRuntimeException. (HLS)
- Moved the code for the Grabber Ant task into the tree and improve the build scripts to dynamically compile and use it. (HLS)
- Typo in jar-module.xml causes broken build if junit library is missing (HLS) Thanks to Johan Lindquist. Fixes HIVEMIND-31.
- Made a number of changes to ensure HiveMind compatibility with JDK 1.3. (HLS) Fixes HIVEMIND-35.
- Changed some unit tests to adapt to platform line endings. (HLS) Fixes HIVEMIND-26.

1.2.2. Version 1.0-beta-1 (Jun 26 2004)

- Added change log. (HLS)
- Refactored ClassFab and related classes for easier reuse outside of HiveMind. Added a new suite of tests related to ClassFab.(HLS)
- Created two new services in hivemind-lib for creating default implementations of arbitrary interfaces (DefaultImplementationBuilder) and for using that to create placeholder services (PlaceholderFactory).(HLS)
- Created MessageFormatter class as a wrapper around ResourceBundle and an easy way for individual packages to gain access to runtime messages. (HLS)
- Modified the read-attribute rule to allow a translator to be specified (overriding the translator for the attribute).(HLS)
- Added the `qualified-id` and `id-list` translators.(HLS)
- Added the hivemind.lib.PipelineFactory and related code, schemas, tests and documentation. (HLS)
- Enhance logging of exceptions when setting a service property to a contribution (HLS) Fixes HIVEMIND-4.

- Added service hivemind.lib.BeanFactoryBuilder. (HLS)
- Removed the <description> element from the module descriptor format; descriptions are now provided as enclosed text for element that support descriptions. (HLS)
- Changed the MethodMatcher classes to use a MethodSignature rather than a Method. (HLS)
- Changed MessageFormatter to automatically convert Throwables into their message or class name. (HLS)
- Added FileResource. (HLS)
- Extended hivemind.BuilderFactory to be able to set the `ClassResolver`; for a service implementation, and to autowire common properties (log, messages, serviceId, errorHandler, classResolver) if the properties are writeable and of the correct type. (HLS)
- Added methods `newControl()`, `newMock()`, `addControl()`, `replayControls()` and `verifyControls()` to `HiveMindTestCase` to simplify test cases that use multiple EasyMock mock objects. (HLS)
- Changed `HiveMindFilter` to log a message after it stores the registry into the servlet context. (HLS)
- Restore the `getConfiguration()` and `expandSymbols()` methods to the `Registry` interface. (HLS) Fixes HIVEMIND-11.
- SimpleDataLanguageParser calls the ContentHandler with a null namespace argument instead of "". That leads to some problems if you want to use transformers. (HLS) Thanks to Dieter Bogdoll. Fixes HIVEMIND-9.
- Fix how certain translator messages are generated to avoid unit test failures. (HLS) Thanks to Achim Hügen. Fixes HIVEMIND-7.
- Modify the build files to enable debugging by default. (HLS) Fixes HIVEMIND-12.
- Added validation of id attributes in module deployment descriptors (using ORO regular expressions). (HLS)
- Fix some typos in definition of the hivemind.lib.NameLookup service. (HLS)
- Fix a mistake in the BuilderFactory's set-object element, and add integration tests. (HLS) Thanks to Naresh Sikha. Fixes HIVEMIND-25.

1.3. Todo List

1.3.1. Release 1.0

- **[lib]** JMX Integration # HLS
- **[project]** Make build scripts more portable. # HLS
- **[documentation]** Update case study #1 #
- **[documentation]** Update interceptor tutorial #

1.4. HiveMind Downloads

HiveMind distributions are available from the Apache Mirrors. HiveMind is packaged somewhat differently than most other Apache projects, in that the main distribution includes binary *and* source, but that documentation is seperate:

- `hivemind-`*release*`.tar.gz` -- Combined binary/source distribution
- `hivemind-`*release*`.zip` -- Combined binary/source distribution (about twice the size of the .tar.gz)
- `hivemind-`*release*`-docs.tar.gz` -- The HiveMind documentation (the same as this site)

Each file also has a MD5 checksum file, so you can verify that what you download is valid, and a GPG key (.asc) to further verify that there has been no tampering.

Note:

Under Internet Explorer, the .tar.gz files do not download with the correct file name. Download them, rename them to .tar.gz and then open them using WinZip.

1.5. CVS Access

Anonymous CVS access is available as:

```
:pserver:anoncvs@cvs.apache.org:/home/cvspublic/jakarta-hivemind
```

In addition, the CVS repository may be browsed online.

1.6. Tutorials and Information

1.6.1. Bootstrapping the Registry

Before you can access the configuration points and services defined in your application's module deployment descriptors, you need a registry; here we'll describe how to construct the registry.

The key class here is RegistryBuilder, which contains code for locating and parsing the module deployment descriptors and constructing a registry from the combined data. The descriptors are all found on the class path; they'll include the descriptors for HiveMind itself with descriptors packaged into your application's JARs.

Note:

As HiveMind grows in popularity, we may start to see third party frameworks come bundled with HiveMind module deployment descriptors ... but it's too soon for that, now.

Let's examine how all this comes together. The layout of the project is shown below.

[Project Layout]

1.6.1.1. Service Interfaces and Implementations

The first step is to define the service interface:

```
package hivemind.examples;

public interface Adder
{
    public int add(int arg0, int arg1);
}
```

Next we need an implementation for that service:

```
package hivemind.examples.impl;

import hivemind.examples.Adder;

public class AdderImpl implements Adder
{

  public int add(int arg0, int arg1)
  {
    return arg0 + arg1;
  }

}
```

The example includes three additional interfaces and matching implementations: for a Subtracter, Multiplier, Divider, and lastly, a Calculator that combines them together:

```
package hivemind.examples;


public interface Calculator extends Adder, Subtracter, Multiplier, Divider
{

}
```

The Calculator implementation will require some wiring; it expects that each of the other four services (Adder, Substracter, Multiplier and Divider) will be plugged into it:

```
package hivemind.examples.impl;

import hivemind.examples.Adder;
import hivemind.examples.Calculator;
import hivemind.examples.Divider;
import hivemind.examples.Multiplier;
import hivemind.examples.Subtracter;

public class CalculatorImpl implements Calculator
{
  private Adder _adder;
  private Subtracter _subtracter;
  private Multiplier _multiplier;
  private Divider _divider;

  public void setAdder(Adder adder)
  {
    _adder = adder;
  }

  public void setDivider(Divider divider)
  {
    _divider = divider;
  }

  public void setMultiplier(Multiplier multiplier)
  {
    _multiplier = multiplier;
  }

  public void setSubtracter(Subtracter subtracter)
  {
    _subtracter = subtracter;
  }

  public int add(int arg0, int arg1)
  {
    return _adder.add(arg0, arg1);
  }

  public int subtract(int arg0, int arg1)
  {
    return _subtracter.subtract(arg0, arg1);
  }

  public int multiply(int arg0, int arg1)
  {
    return _multiplier.multiply(arg0, arg1);
  }

  public int divide(int arg0, int arg1)
  {
    return _divider.divide(arg0, arg1);
  }
}
```

1.6.1.2. Module Deployment Descriptor

Finally, we need the HiveMind module deployment descriptor, `hivemodule.sdl`. This file is in Simple Data Language format (though equivalent XML is supported if the file is named `hivemodule.xml`).

The module descriptor creates each of the services in terms of an interface, and an implementation. In addition, each service gets its own logging interceptor.

```
module (id=hivemind.examples version="1.0.0")
{
  service-point (id=Adder interface=hivemind.examples.Adder)
  {
    create-instance (class=hivemind.examples.impl.AdderImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Subtracter interface=hivemind.examples.Subtracter)
  {
    create-instance (class=hivemind.examples.impl.SubtracterImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Multiplier interface=hivemind.examples.Multiplier)
  {
    create-instance (class=hivemind.examples.impl.MultiplierImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Divider interface=hivemind.examples.Divider)
  {
    create-instance (class=hivemind.examples.impl.DividerImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Calculator interface=hivemind.examples.Calculator)
  {
    invoke-factory (service-id=hivemind.BuilderFactory)
    {
      construct (class=hivemind.examples.impl.CalculatorImpl)
      {
        set-service (property=adder service-id=Adder)
        set-service (property=subtracter service-id=Subtracter)
        set-service (property=multiplier service-id=Multiplier)
        set-service (property=divider service-id=Divider)
      }
    }

    interceptor (service-id=hivemind.LoggingInterceptor)
  }
}
```

Here we've chosen to have the module id, `hivemind.examples`, match the package name but that is not an absolute requirement.

The interesting part is the use of the hivemind.BuilderFactory to construct the Calculator service and connect it to the other four services.

1.6.1.3. Building the Registry

Before your code can access any services (or configuration points), it must construct the Registry. The Registry is the applications gateway into the services and configurations managed by HiveMind.

```
package hivemind.examples;

import org.apache.hivemind.Registry;
import org.apache.hivemind.impl.RegistryBuilder;

public class Main
{

  public static void main(String[] args)
  {
    int arg0 = Integer.parseInt(args[0]);
    int arg1 = Integer.parseInt(args[1]);

    Registry registry = RegistryBuilder.constructDefaultRegistry();

    Calculator c =
      (Calculator) registry.getService("hivemind.examples.Calculator", Calculator.class

    System.out.println("Inputs " + arg0 + " and " + arg1);

    System.out.println("Add     : " + c.add(arg0, arg1));
    System.out.println("Subtract: " + c.subtract(arg0, arg1));
    System.out.println("Multiply: " + c.multiply(arg0, arg1));
    System.out.println("Divide  : " + c.divide(arg0, arg1));
  }
}
```

RegistryBuilder contains a static method for constructing a Registry, which is suitable for most situations.

Now that we have the registry, we can use the fully qualified id of the Calculator service, `hivemind.examples.Calculator`, to get the service implementation. We pass in the class that we'll be casting the service to ... this allows HiveMind to produce a more meaningful error than a ClassCastException.

Using the reference to the Calculator service, we can finally invoke the `add()`, `subtract()`, `multiply()` and `divide()` methods.

1.6.1.4. Building the Example

Building and running the example using Ant is a snap; all the details are in the `build.xml`:

```
<?xml version="1.0"?>

<project name="HiveMind Adder Example" default="jar">

  <property name="java.src.dir" value="src/java"/>
  <property name="test.src.dir" value="src/test"/>
  <property name="conf.dir" value="src/conf"/>
  <property name="descriptor.dir" value="src/descriptor"/>
  <property name="target.dir" value="target"/>
  <property name="classes.dir" value="${target.dir}/classes"/>
  <property name="test.classes.dir" value="${target.dir}/test-classes"/>
  <property name="example.jar" value="${target.dir}/hivemind-examples.jar"/>
  <property name="lib.dir" value="lib"/>
  <property name="junit.temp.dir" value="${target.dir}/junit-temp"/>
  <property name="junit.reports.dir" value="${target.dir}/junit-reports"/>

  <path id="build.class.path">
    <fileset dir="${lib.dir}">
      <include name="*.jar"/>
    </fileset>
  </path>

  <path id="test.build.class.path">
    <path refid="build.class.path"/>
    <path location="${classes.dir}"/>
  </path>

  <path id="run.class.path">
    <path refid="build.class.path"/>
    <pathelement location="${classes.dir}"/>
    <pathelement location="${descriptor.dir}"/>
    <pathelement location="${conf.dir}"/>
  </path>

  <path id="test.run.class.path">
    <path refid="run.class.path"/>
    <path location="${test.classes.dir}"/>
  </path>

  <target name="clean" description="Delete all derived files.">
    <delete dir="${target.dir}" quiet="true"/>
  </target>

  <target name="compile" description="Compile all Java code.">
    <mkdir dir="${classes.dir}"/>
    <javac srcdir="${java.src.dir}" destdir="${classes.dir}" classpathref="build.class.
  </target>
  <target name="compile-tests" description="Compile test classes." depends="compile">
    <mkdir dir="${test.classes.dir}"/>
    <javac srcdir="${test.src.dir}" destdir="${test.classes.dir}" classpathref="test.bu
  </target>

  <target name="run-tests" description="Run unit tests." depends="compile-tests">

    <mkdir dir="${junit.temp.dir}"/>
    <mkdir dir="${junit.reports.dir}"/>

    <junit haltonfailure="off" failureproperty="junit-failure" tempdir="${junit.temp.di
      <classpath refid="test.run.class.path"/>

      <formatter type="xml"/>
      <formatter type="plain"/>
      <formatter type="brief" usefile="false"/>

      <batchtest todir="${junit.reports.dir}">
        <fileset dir="${test.classes.dir}">
          <include name="**/Test*.class"/>
        </fileset>
      </batchtest>
    </junit>

    <fail if="junit-failure" message="Some tests failed."/>

  </target>

  <target name="jar" description="Construct the JAR file." depends="compile,run-tests">
    <jar destfile="${example.jar}">
      <fileset dir="${classes.dir}"/>
    <fileset dir="${descriptor.dir}"/>
    </jar>
  </target>

  <target name="run" depends="compile" description="Run the Adder service.">
    <java classname="hivemind.examples.Main" classpathref="run.class.path" fork="true">
      <arg value="11"/>
      <arg value="23"/>
    </java>
  </target>

</project>
```

The important part is to package both the classes and the HiveMind module deployment descriptor into the JAR.

The only other oddity was to add `src/conf` to the runtime classpath; this is to include the `log4j.properties` configuration file; otherwise Log4J will write console errors about missing configuration.

1.6.1.5. Running the Examples

```
bash-2.05b$ ant run
Buildfile: build.xml

compile:
    [mkdir] Created dir: C:\workspace\hivemind-example\target\classes
    [javac] Compiling 15 source files to C:\workspace\hivemind-example\target\classes

run:
     [java] Inputs 11 and 23
     [java] Add     : 34
     [java] Subtract: -12
     [java] Multiply: 253
     [java] Divide  : 0



BUILD SUCCESSFUL
Total time: 3 seconds
```

1.6.2. Inversion of Control

Seems like **Inversion of Control** is all the rage these days. The Avalon project is completely based around it. Avalon uses detailed assembly descriptions to tie services together ... there's no way an Avalon component can "look up" another component; in Avalon you explicitly connect services together.

That's the basic concept of Inversion of Control; you don't create your objects, you describe how they should be created. You don't directly connect your components and services together in code, you describe which services are needed by which components, and the container is responsible for hooking it all together. The container creates all the objects, wires them together by setting the necessary properties, and determines when methods are invoked.

More recently, this concept has been renamed Dependency Injection.

There are three different implementation pattern types for IoC:

type-1	Services need to implement a dedicated interface through which they are provided with an object from which they can look up dependencies (other services). This is the pattern used by the earlier containers provided by Avalon.
type-2	Services dependencies upon are assigned via JavaBeans properties (setter methods). Both HiveMind and Spring use this approach.
type-3	Services dependencies are provided as constructor parameters (and are not exposed as

	JavaBeans properties). This is the exclusive approach used by PicoContainer, and is also used in HiveMind and Spring.

HiveMind is a much looser system than Avalon. HiveMind doesn't have an explicit assembly stage; it wires together all the modules it can find at runtime. HiveMind is responsible for creating services (including core implementations and interceptors). It is quite possible to create service factories that do very container-like things, including connecting services together. hivemind.BuilderFactory does just that, instantiating an object to act as the core service implementation, then setting properties of the object, some of which are references to services and configuration point element data.

In HiveMind, you are free to mix and match type-2 (property injection) and type-3 (constructor injection), setting some (or all) dependencies via a constructor and some (or all) via JavaBeans properties.

In addition, JavaBeans properties (for dependencies) can be write-only. You only need to provide a setter method. The properties are properties of the core service implementation, there is no need for the accessor methods to be part of the service interface.

HiveMind's lifecycle support is much more rudimentary than Avalon's. Your service implementations can get hivemindcallbacks when they are first created, and when they are discarded, by implementing certain interfaces.

Purist inversion of control, as in Avalon, may be more appropriate in well-constrained systems containing untrusted code. HiveMind is a layer below that, not an application server, but a microkernel. Although I can see using HiveMind as the infrastructure of an application server, even an Avalon application server, it doesn't directly overlap otherwise.

1.6.3. HiveMind Localization

Every HiveMind module may have its own set of messages. Messages are stored alongside the module deployment descriptor, as `META-INF/hivemodule.properties` (within the module's JAR).

Note:

In actuality, the name of the properties file is created by stripping off the extension (".sdl" or ".xml") from the descriptor name and appending the localization code and ".properties". This is relevant only if you load your module deployment descriptors from a non-standard location, possibly via the <sub-module> element.

Services can gain access to localized messages, as an instance of Messages , which includes methods for accessing messages and formatting messages with arguments.

In a module descriptor, within the <contribution> and <invoke-factory> elements, you can reference a localized message in an attribute or element content simply by prefixing the message key with '%'. Examples:

```
contribution (configuration-id=...)
{
  some-item (message="%message.key")
  {
    "%other.message.key"
  }
}
```

The two keys (`message.key` and `other.message.key`) are searched for in the *contributing* module's messages.

HiveMind gracefully recovers from undefined messages. If a message is not in the properties file, then HiveMind provides a substitute value by converting the key to upper-case and adding brackets, i.e. `[MESSAGE.KEY]`. This allows your application to continue running, but clearly identifies missing messages.

By adding additional files, message localization can be accomplished. For example, adding a second file, `META-INF/hivemodule_fr.properties` would provide French language localizations. Any common keys between the two files defer to the more specific file.

1.6.3.1. Setting the locale

When a Registry is created by the RegistryBuilder, a locale is specified. This is the locale for the Registry and, by extension, for all Modules in the registry. The locale may not be changed.

1.6.4. HiveMind Multi-Threading

HiveMind is specifically targetted for J2EE: deployment in a WAR or EAR, particularly as part of a Tapestry application. Of course, J2EE is not a requirement, and HiveMind is quite useful even in a simple, standalone environment.

In the world of J2EE, multi-threading is always an issue. HiveMind services are usually singletons, and must be prepared to operate in a multi-threaded environment. That means services should not have any specific state, much like a servlet.

1.6.4.1. Construction State

HiveMind expects that initially, work will progress in a single startup thread. This is the early state, the construction state, where the module deployment descriptors are located and parsed, and the contents used to assemble the registry; this is the domain of RegistryBuilder .

The construction activities are not thread-safe. This includes the parser, and other code (virtually all of which is hidden from your application).

The construction state ends when the `RegistryBuilder` returns the Registry from method `constructRegistry()`. The registry is thread-safe.

1.6.4.2. Runtime State

Everything that occurs with the Registry and modules must be thread-safe. Key methods are always synchronized. In particular, the methods that construct a service and construct configuration point elements are thread-safe. Operations such as building the interceptor stack, instantiating core service implementations, and converting XML to Java objects operate in a thread-safe manner. However, different threads may be building different services simultaneously. This means that, for example, an interceptor service implementation must still be thread-safe, since it may be called upon to generate interceptors for two or more different services simultaneously.

On the other hand, the Java objects constructed from XML <rules> don't need to be thread-safe, since that construction is synchronized properly ... only a single thread will be converting XML to Java objects for any single configuration point.

1.6.4.3. Managing Service State

When services simply must maintain state *between* method invocations, there are several good options:

- Store the data in an object passed to or returned from the service
- Make use of the hivemind.ThreadLocalStorage service to store the data in a thread-local map.
- Make use of the threaded or pooled service models, which allow a service to keep its state between service method invocations.

1.6.5. HiveMind Servlet Filter

HiveMind includes a feature to streamline the use of HiveMind within a web application: a servlet filter that can automatically construct the HiveMind Registry and ensure that end-of-request thread cleanup occurs.

The filter class is HiveMindFilter. It constructs a standard HiveMind Registry when initialized, and will shutdown the Registry when the containing application is undeployed.

Each request will be terminated with a call to the Registry's cleanupThread() method, which will cleanup any thread-local values, including service implementations that are bound to the current thread.

The HiveMindFilter class includes a static method for accessing the Registry.

1.6.5.1. Deployment Descriptor

To make use of the filter, it must be declared inside the web deployment descriptor (web.xml). Filters can be attached to servlets, or URL patterns, or both. Here's an example:

```
<filter>
  <filter-name>HiveMindFilter</filter-name>
  <filter-class>org.apache.hivemind.servlet.HiveMindFilter</filter-class>
</filter>

<servlet>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>myco.servlets.MyServlet</servlet-class>
</servlet>

<filter-mapping>
  <filter-name>HiveMindFilter</filter-name>
  <servlet-name>MyServlet</servlet-name>
</filter-mapping>
```

1.6.6. Creating New Interceptors

Interceptors are used to add behavior to a HiveMind service after the fact. An interceptor sits between the client code and the core service implementation; it implements the service interface. For each method in the service interface, the interceptor will re-invoke the method on the next object in the chain ... either another interceptor, or the core service implementation.

That's not useful ... but when the interceptor does something before and/or after re-invoking the method, it can easily add quite a bit of useful, robust functionality.

In fact, if you've heard about "Aspect Oriented Programming", interceptors are simply one kind of aspect, a method introduction, based on the service interface.

Be warned; interceptors are an example of programs writing other programs; it's a whole new level of abstraction and requires a bit of getting used to. Also, note that the term "interceptor" can mean two different, related things: a service interceptor factory or a fabricated class created by the factory; this should be obvious by the context.

1.6.6.1. Interceptor Factories

Interceptors are created, at runtime, by service interceptor factories. A service interceptor factory builds a custom class at runtime using the Javassist library. The class is then instantiated.

Note:

The use of Javassist is not mandated but is generally easy and is more efficient at runtime. It is possible to accomplish the same thing using JDK proxies.

Interceptor factories are HiveMind services which implement the ServiceInterceptorFactory interface. This interface has a single method, `createInterceptor()` , which is passed:

- The InterceptorStack (an object used to manage the process of creating interceptors for a service)
- The Module which invoked the interceptor factory
- A list of parameters

Like service implementation factories, interceptor factories may take parameters; they may identify a <schema> which is used to convert any XML enclosed by the <interceptor> element into Java objects. Many interesting interceptors can be created without needing parameters to guide the fabrication of the interceptor class.

1.6.6.2. Implementing the NullInterceptor

To demonstrate how easy it is to create an interceptor, we'll start with a NullInterceptor. NullInterceptor does not add any functionality, it simply re-invokes each method on its *inner*. The *inner* is the next interceptor, or the core service implementation ... an interceptor doesn't know or care which.

Simple interceptors, those which do not take any parameters, are implemented by subclassing AbstractServiceInterceptorFactory. It does most of the work, organizing the process of creating the class and methods ... even adding a `toString()` method implementation automatically.

NullInterceptor Class

Most of the work for creating a standard service interceptor factory is taken care of by the AbstractServiceInterceptorFactory base class. All that's left is to define what happens for each method in the service interface.

```
package com.example.impl;
```

```
import java.lang.reflect.Modifier;

import org.apache.hivemind.service.ClassFab;
import org.apache.hivemind.service.impl.AbstractServiceInterceptorFactory;

public class NullInterceptor extends AbstractServiceInterceptorFactory
{

  protected void addServiceMethodImplementation(
    ClassFab classFab,
    String methodName,
    Class returnType,
    Class[] parameterTypes,
    Class[] exceptionTypes)
  {
    classFab.addMethod(
      Modifier.PUBLIC,
      methodName,
      returnType,
      parameterTypes,
      exceptionTypes,
       "{ return ($r) _inner." + methodName + "($$); }");
  }
}
```

The `addServiceMethodImplementation()` method is invoked for each service method. It is passed the ClassFab, an object which represents a class being fabricated, which allows new fields, methods and constructors to be added.

ClassFab and friends are just a wrapper around the Javassist framework, a library used for runtime bytecode enhancement and other aspect oriented programming tasks. HiveMind uses only a small fraction of the capabilities of Javassist. Javassist's greatest feature is how new code is specified ... it looks like ordinary Java source code, with a few additions.

The `_inner` variable is a private instance variable, the inner for this interceptor. The `($r)` reference means "cast to the return type for this method", and properly handles void methods. The `$$` is a placeholder for a comma-seperated list of all the parameters to the method.

Put together, this simply says "reinvoke the method on the next instance."

AbstractServiceInterceptorFactory is responsible for creating the `_inner` variable and building the constructor which sets it up, as well as invoking the constructor on the completed interceptor class.

Declaring the Service

To use a service, it is necessary to declare the service in a module deployment descriptor. The AbstractServiceInterceptorFactory base class expects two properties to be set when the service is constructed, `serviceId` and `factory`:

```
service-point (id=NullInterceptor interface=org.apache.hivemind.ServiceInterceptorFacto
{
  invoke-factory (service-id=hivemind.BuilderFactory)
  {
    construct (class=com.example.impl.NullInterceptor service-id-property=serviceId)
    {
      set-service (property=factory service-id=hivemind.ClassFactory)
    }
  }
}
```

1.6.6.3. Implementing the hivemind.LoggingInterceptor service

A more involved example is the LoggingInterceptor service, which adds logging capabilities to services. It's a bit more involved than NullInterceptor, and so overrides more methods of AbstractServiceInterceptorFactory.

Note:
The logging interceptor has changed recently, to allow flexibilty (via parameters) on which methods are logged. This document has not yet been updated.

AbstractLoggingInterceptor base class

In most cases, an abstract base class for the interceptor is provided; in this case, it is AbstractLoggingInterceptor. This class provides several protected methods used by fabricated interceptors. To help ensure that there are no conflicts between the method of the service interface and the methods provided by the super-class, the provided methods are named with a leading underscore. These methods are:

- `_logEntry()` to log entry to a method
- `_logExit()` to log exit from a method, with return value
- `_logVoidExit()` to log exit from a void method (no return value)
- `_logException()` to log an exception thrown when the method is executed
- `_isDebugEnabled()` to determine if debugging is enabled or disabled

In addition, there's a protected constructor, which takes an instance of `org.apache.commons.logging.Log` that must be invoked from the fabricated subclass.

Method `getInterceptorSuperclass()` is used to tell AbstractServiceInterceptorFactory which class to use as the base:

```
protected Class getInterceptorSuperclass() { return
                                        AbstractLoggingInterceptor.class; }
```

Creating the infrastructure

The method `createInfrastructure()` is used to add fields and constructors to the interceptor class.

```
protected void createInfrastructure(InterceptorStack stack, ClassFab classFab)
{
  Class topClass = stack.peek().getClass();

  classFab.addField("_inner", topClass);

  classFab.addConstructor( new Class[] { Log.class, topClass },
    null,
    "{ super($1); _inner = $2; }");
}
```

Since, when a interceptor is created, the inner object has already been created, we can use its *actual type* for the `_inner` field. This results in a much more efficient method invocation than if `_inner`'s type was the service interface.

Instantiating the Instance

The method `instantiateInterceptor()` is used to create a new instance from the fully fabricated class.

```
protected Object instantiateInterceptor(InterceptorStack stack, Class interceptorClass)
{
  Object stackTop = stack.peek();
  Class topClass = stackTop.getClass();
  Log log = LogFactory.getLog(stack.getServiceExtensionPointId());

  Constructor c =  interceptorClass.getConstructor(new Class[] { Log.class, topClass })

  return c.newInstance(new Object[] { log, stackTop });
}
```

This implementation gets the top object from the stack (the inner object for this interceptor) and the correct `Log` instance (based on the service extension point id ... for the service being extended with the interceptor). The constructor, created by `createInfrastructure()` is accessed and invoked to create the interceptor.

Adding the Service Methods

The last, and most complex, part of this is the method which actually creates each service method.

```
  protected void addServiceMethodImplementation(
    ClassFab classFab,
    String methodName,
    Class returnType,
    Class[] parameterTypes,
    Class[] exceptions)
  {
    boolean isVoid = (returnType == void.class);

    BodyBuilder builder = new BodyBuilder();

    builder.begin();
    builder.addln("boolean debug = _isDebugEnabled();");

    builder.addln("if (debug)");
    builder.add("  _logEntry(");
    builder.addQuoted(methodName);
    builder.addln(", $args);");

    if (!isVoid)
    {
      builder.add(ClassFabUtils.getJavaClassName(returnType));
      builder.add(" result = ");
    }

    builder.add("_inner.");
    builder.add(methodName);
    builder.addln("($$);");

    if (isVoid)
    {
      builder.addln("if (debug)");
      builder.add("  _logVoidExit(");
      builder.addQuoted(methodName);
      builder.addln(");");
    }
    else
    {
      builder.addln("if (debug)");
      builder.add("  _logExit(");
      builder.addQuoted(methodName);
      builder.addln(", ($w)result);");
      builder.addln("return result;");
    }

    builder.end();

    MethodFab methodFab =
      classFab.addMethod(
        Modifier.PUBLIC,
        methodName,
        returnType,
        parameterTypes,
        exceptions,
        builder.toString());

    builder.clear();
    builder.begin();
    builder.add("_logException(");
    builder.addQuoted(methodName);
    builder.addln(", $e);");
    builder.addln("throw $e;");
    builder.end();

    String body = builder.toString();

    int count = exceptions == null ? 0 : exceptions.length;

    for (int i = 0; i < count; i++)
    {
      methodFab.addCatch(exceptions[i], body);
    }

    // Catch and log any runtime exceptions, in addition to the
    // checked exceptions.

    methodFab.addCatch(RuntimeException.class, body);
  }
```

A bit more is going on here; since the method bodies for the fabricated methods are more complex, we're using the BodyBuilder class to help assemble them (but still keep the final method body neat and readable). BodyBuilder's begin() and end() methods take care of open and close braces, as well as indentation inside a code block.

When you implement logging in your own classes, you often invoke the method `Log.isDebugEnabled()` multiple times ... but in the fabricated class, the method is only invoked once and cached for the duration of the call ... a little efficiency gained back.

Likewise, if a method can throw an exception or return from the middle, its hard to be assured that you've logged every exit, or overy thrown exception; taking this code out into an interceptor class ensures that its done consistently and properly.

1.6.6.4. Implementing Interceptors with Parameters

Interceptor factories may take parameters ... but then their implementation can't be based on AbstractServiceInterceptorFactory. The hivemind.LoggingInterceptor is an example of such a factory (its parameters determine which methods do and don't get logging). The basic approach is the same ... you just need a little extra work to validate, interpret and use the parameters.

When would such as thing be useful? One example is declarative security; you could specify, on a method-by-method basis, which methods were restricted to which roles.

1.6.6.5. Conclusion

Interceptors are a powerful concept that allow you to add consistent, efficient, robust behavior to your services. It takes a little while to wrap your brain around the idea of classes writing the code for other classes ... but once you do, a whole world of advanced techniques opens up to you!

1.6.7. Overriding a Service

It is not uncommon to want to override an existing service and replace it with a new implementation. This goes beyond simply intercepting the service ... the goal is to replace the original implementation with a new implementation. This occurs frequently in Tapestry where frequently an existing service is replaced with a new implementation that handles application-specific cases (and delegates most cases to the default implementation).

Note:

Plans are afoot to refactor Tapestry 3.1 to make considerable use of HiveMind. Many of the ideas represented in HiveMind germinated in earlier Tapestry releases.

HiveMind doesn't have an explicit mechanism for accomplishing this ... that's because its reasonable to replace and wrap existing services just with the mechanisms already available.

1.6.7.1. Step One: A non-overridable service

To describe this technique, we'll start with a ordinary, every day service. In fact, for discussion purposes, there will be two services: Consumer and Provider. Ultimately, we'll show how to override Provider. Also for discussion purposes, we'll do all of this in a single module, though (of course) you can as easily split it up across many modules.

To begin, we'll define the two services, and set Provider as a property of Consumer:

```
module (id=ex.override version="1.0.0")
{
  service-point (id=Provider interface=ex.override.Provider)
  {
    create-instance (class=ex.override.impl.ProviderImpl)
  }

  service-point (id=Consumer interface=ex.override.Consumer)
  {
    invoke-factory (service-id=hivemind.BuilderFactory)
    {
      construct (class=ex.override.impl.Consumer)
      {
        set-service (property=provider service-id=Provider)
      }
    }
  }
}
```

1.6.7.2. Step Two: Add some indirection

In this step, we still have just the two services ... Consumer and Provider, but they are linked together less explicitly, by using substitution symbols.

```
module (id=ex.override version="1.0.0")
{
  service-point (id=Provider interface=ex.override.Provider)
  {
    create-instance (class=ex.override.impl.ProviderImpl)
  }

  contribution (configuration-id=hivemind.FactoryDefaults)
  {
    default (symbol=ex.override.Provider value=ex.override.Provider)
  }

  service-point (id=Consumer interface=ex.override.consumer)
  {
    invoke-factory (service-id=hivemind.BuilderFactory)
    {
      construct (class=ex.override.impl.Consumer)
      {
        set-service (property=provider service-id=${ex.override.Provider})
      }
    }
  }
}
```

The end result is the same ... the symbol `ex.override.Provider` evaluates to the service id `ex.override.Provider` and the end result is the same as step one. We needed to use a fully qualified service id because, ultimately, we don't know in which modules the symbol will be referenced.

1.6.7.3. Step Three: Override!

The final step is to define a second service and slip it into place. For kicks, the OverrideProvider service will get a reference to the original Provider service.

```
module (id=ex.override version="1.0.0")
{
  service-point (id=Provider interface=ex.override.Provider)
  {
    create-instance (class=ex.override.impl.ProviderImpl)
  }

  contribution (configuration-id=hivemind.FactoryDefaults)
  {
    default (symbol=ex.override.Provider value=ex.override.Provider)
  }

  service-point (id=OverrideProvider interface=ex.override.Provider)
  {
    invoke-factory (service-id=hivemind.BuilderFactory)
    {
      construct (class=ex.override.impl.OverrideProviderImpl)
      {
        set-service (property=defaultProvider service-id=Provider)
      }
    }
  }

  // ApplicationDefaults overrides FactoryDefaults

  contribution (configuration-id=hivemind.ApplicationDefaults)
  {
          // Must specify the fully qualified service id (the symbol
          // may be evaluated in an unknown context later)

    default (symbol=ex.override.Provider value=ex.override.OverrideProvider)
  }

  service-point (id=Consumer interface=ex.override.consumer)
  {
    invoke-factory (service-id=hivemind.BuilderFactory)
    {
      construct (class=ex.override.impl.Consumer)
      {
        set-service (property=provider service-id=${ex.override.Provider})
      }
    }
  }
}
```

The new service, OverrideProvider, gets a reference to the original service using its real id. It can't use the symbol that the Consumer service uses, because that would end up pointing it at itself. Again, in this example it's all happening in a single module, but it could absolutely be split up, with OverrideProvider and the configuration to hivemind.ApplicationDefaults in an entirely different module.

hivemind.ApplicationDefaults overrides hivemind.FactoryDefaults. This means that the Consumer will be connected to `ex.override.OverrideProvider`.

Note that the <service-point> for the Consumer doesn't change between steps two and three.

1.6.7.4. Limitations

The main limitation to this approach is that you can only do it once for a service; there's no way to add an EvenMoreOverridenProvider service that wraps around OverrideProvider (that wraps around Provider). Making multiple contributions to the hivemind.ApplicationDefaults configuration point with the name symbol name will result in a runtime error ... and unpredictable results.

This could be addressed by adding another source to the hivemind.SymbolSources configuration.

To be honest, if this kind of indirection becomes extremely frequent, then HiveMind should change to accomidate the pattern, perhaps adding an `<override>` element, similar to a <interceptor> element.

1.7. Reports

1.7.1. Project License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.
      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution."

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      copyright license to reproduce, prepare Derivative Works of,
      publicly display, publicly perform, sublicense, and distribute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have made,
      use, offer to sell, sell, import, and otherwise transfer the Work,
      where such license applies only to those patent claims licensable
      by such Contributor that are necessarily infringed by their
      Contribution(s) alone or by combination of their Contribution(s)
      with the Work to which such Contribution(s) was submitted. If You
      institute patent litigation against any entity (including a
      cross-claim or counterclaim in a lawsuit) alleging that the Work
      or a Contribution incorporated within the Work constitutes direct
      or contributory patent infringement, then any patent licenses
      granted to You under this License for that Work shall terminate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent notices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative Works
          that You distribute, all copyright, patent, trademark, and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any part of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of its
          distribution, then any Derivative Works that You distribute must
          include a readable copy of the attribution notices contained
          within such NOTICE file, excluding those notices that do not
          pertain to any part of the Derivative Works, in at least one
          of the following places: within a NOTICE text file distributed
          as part of the Derivative Works; within the Source form or
          documentation, if provided along with the Derivative Works; or,
          within a display generated by the Derivative Works, if and
          wherever such third-party notices normally appear. The contents
          of the NOTICE file are for informational purposes only and
          do not modify the License. You may add Your own attribution
          notices within Derivative Works that You distribute, alongside
          or as an addendum to the NOTICE text from the Work, provided
          that such additional attribution notices cannot be construed
          as modifying the License.

      You may add Your own copyright statement to Your modifications and
      may provide additional or different license terms and conditions
      for use, reproduction, or distribution of Your modifications, or
      for any such Derivative Works as a whole, provided Your use,
      reproduction, and distribution of the Work otherwise complies with
      the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state otherwise,
      any Contribution intentionally submitted for inclusion in the Work
      by You to the Licensor shall be under the terms and conditions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede or modify
      the terms of any separate license agreement you may have executed
      with Licensor regarding such Contributions.
```

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

```
   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
```

1.7.2.

2. Module: hivemind

2.1. Services

2.1.1. hivemind.BuilderFactory Service

The BuilderFactory service is a service implementation factory ... a service that is used to construct other services.

The general usage of the BuilderFactory is:

```
invoke-factory (service-id=hivemind.BuilderFactory)
{
  construct (class=... autowire-services=... log-property=... messages-property=...
    service-id-property=... initialize-method=...
    error-handler-property=... class-resolver-property=...)
  {
    log
    messages
    service-id
    error-handler
    class-resolver
    string { ... }
    boolean { ... }
    configuration { ... }
    int { ... }
    long { ... }
    resource { ... }
    service { ... }
    object { ... }

    event-listener (service-id=... event-site-name=...)
    set (property=... value=...)
    set-configuration (property=... configuration-id=...)
    set-resource (property=... path=...)
    set-service (property=... service-id=...)
    set-object (property=... value=...)
  }
}
```

The attributes of the `construct` element are used to specify the implementation class and set common service properties. Nested elements supply the constructor parameters and configure other specific properties of the implementation (the `set-...` elements).

> **Note:**
> BuilderFactory is a complex tool, with support for both constructor dependency injection and property dependency injection. Many of the options are rarely used; the most general purpose and most frequently used are set, set-object and event-listener (along with autowiring of certain properties).

2.1.1.1. construct

Attribute	Required ?	Description
autowire-services	no	If true (the default), then the BuilderFactory will attempt to automatically wire any services that are not otherwise set. Any property that is writable, and whose type is an interface, will be autowired. For such properties, it is required that there be a single service point that implements the interface. An error will be logged if no service point implements the interface, or if multiple service points implement the interface.
class	yes	The fully qualified name of the class to instantiate.
class-resolver-property	no	The property to receive the module's ClassResolver.
error-handler-property	no	The name of a property to recieve the module's ErrorHandler instance (which is used to report recoverable errors).
initialize-method	no	The name of a method (public, no parameters) to invoke after

		the service is constructed, to allow it to perform any final initializion before being put into use.
log-property	no	The name of a property which will be assigned a `org.apache.commons.logging.Log` instance for the service. The Log is created from the complete service id (not the name of the class). If ommitted, no Log will be assigned.
messages-property	no	Allows the [Messages]() for the module to be assigned to a property of the instance.
service-id-property	no	Allows the service id of the *constructed* service to be assigned to a property of the service implementation.

The remaining elements are enclosed by the <construct> element, and are used to supply constructor parameters and configure properties of the constructed service implementation.

2.1.1.2. Autowiring

BuilderFactory will automatically set certain common properties of the service implementation. By using standard names (and standard types), the need to specify attributes `log-property`, `error-handler-property`, etc. is avoided. Simply by having a writable property with the correct name and type is sufficient:

Property name	Property Type
classResolver	[ClassResolver]()
errorHandler	[ErrorHandler]()
log	`org.apache.commons.logging.Log`
messages	[Messages]()
serviceId	String

In addition, if the `initialize-method` attribute is *not* specified, and the service implementation includes a public method `initializeService()` (no parameters, returns void), then `initializeService()` will be invoked as the initializer.

2.1.1.3. Constructor Parameter Elements

The following table summarizes the elements which can be used to specify constructor parameters for the class to instantiate. These elements can be mixed freely with the properties configuring elements. It is important to know that the number, type, and order of the constructor parameter elements determine the constructor that will be used to instantiate the implementation.

Element	Matched Parameter Type	Passed Parameter Value
error-handler	ErrorHandler	The module's ErrorHandler, user to report recoverable errors.
log	org.apache.commons.logging.Log	The Log is created from the complete service id (not the name of the class) of the created service.
messages	org.apache.hivemind.Messages	The Messages object of the invoking module.
object	*variable*	As determined by the object translator, this is decidedly free-form. See hivemind.ObjectProviders.
service-id	java.lang.String	The service id of the *constructed* service.
string	java.lang.String	This element's content.
boolean	boolean	This element's content. Must be either "true" or "false".
configuration	java.util.List	The List of the elements of the configuration specified by this element's content as a configuration id. The id can either by a simple id for a configuration within the same module as the constructed service, or a complete id.
int	int	This element's content parsed as an integer value.
long	long	This element's content parsed

		as a long value.
resource	org.apache.hivemind.Resource	This element's content parsed as a path to a Resource, which is relative to the contributing module's deployment descriptor. If available, a localized version of the Resource will be selected.
service	interface corresponding to specified service	The implementation of the service with the id given in this element's content. The id can either be a simple id for a service within the same module as the constructed service, or a complete id.

2.1.1.4. Service Property Configuring Elements

event-listener

Attribute	Description
service-id	The service which produces events. The service must provide, in its service interface, the necessary add and remove listener methods.
name	The name of an event set to be registered. If not specified, all applicable event sets are used.

If the name attribute is not specified, then BuilderFactory will register for all applicable event sets. For each event set provided by the specified service, BuilderFactory will check to see if the service instance being constructed implements the corresponding listener interface ... if so, the constructed service instance is added as a listener. When the name attribute is specified, the constructed service instance is registered as a listener of just that single type.

Event notifications go directly to the constructed service instance; they don't go through any proxies or interceptors for the service. The service *instance* must implement the listener interface, the constructed service's service interface *does not* have to extend the listener interface. In other words, event notifications are "behind the scenes", not part of the public API of the service.

It is perfectly acceptible to include multiple <event-listener> elements for a number of different event producing services.

It is not enough for the event producer service to have an add listener method (i.e., `addPropertyChangeListener(PropertyChangeListener)`). To be recognized as an event set, there must also be a corresponding remove listener method (i.e., `removePropertyChangeListener(PropertyChangeListener)`), even though BuilderFactory does not make use of the remove method. This is an offshoot of how the JavaBeans API defines event sets.

set

Attribute	Description
property	The name of the property to set.
value	A value to assigned to the property. The value will be converted to an appropriate type for the property.

set-configuration

Attribute	Description
property	The name of the property to set.
configuration-id	The id of a configuration, either a simple id for a configuration within the same module as the constructed service, or a complete id. The property will be assigned a `List` of the elements of the configuration.

set-object

Attribute	Description
property	The name of the property to set.
value	The *selector* used to find an object value. The selector consists of a prefix (such as "service" or "configuration"), a colon, and a *locator* whose interpretation is defined by the prefix. For example, `service:MyService`. See hivemind.ObjectProviders.

set-resource

Attribute	Description
property	The name of the property to set.

path	The path to a Resource, relative to the contributing module's deployment descriptor. If available, a localized version of the Resource will be selected.

set-service

Attribute	Description
property	The name of the property to set.
service-id	The id of a service, either a simple id for a service within the same module as the constructed service, or a complete id. The property will be assigned the service.

2.1.2. hivemind.LoggingInterceptor Service

The LoggingInterceptor service is used to add logging capability to a service, i.e.:

```
interceptor (service-id=hivemind.LoggingInterceptor)
{
  include (method=...)
  exclude (method=...)
}
```

The service make take parameters (which control which methods will be logged).

The logging interceptor uses a `Log` derived from the service id (of the service to which logging is being added).

The service logs, at debug level, the following events:

- Method entry (with parameters)
- Method exit (with return value, if applicable)
- Thrown exceptions (checked and runtime)

By default, the interceptor will log all methods. By supplying parameters to the interceptor, you can control exactly which methods should be logged. The `include` and `exclude` parameter elements specify methods to be included (logged) and excluded (not logged). The `method` attribute is a *method pattern*, a string used to match methods based on name, number of parameters, or type of parameters; see the MethodMatcher class for more details.

A method which does not match any supplied pattern *will* be logged.

2.1.3. hivemind.ShutdownCoordinator Service

Service implementations that need to perform any special shutdown logic should implement the RegistryShutdownListener interface, and let thehivemind.BuilderFactory register them for notifications.

2.1.4. hivemind.ThreadLocalStorage Service

The ThreadLocalStorage service implements the ThreadLocalStorage interface. This service acts as a kind of Map for temporary data. The map is local to the current thread, and is cleared at the end of the transaction.

It is your responsibility to ensure that keys are unique, typically by prefixing them with a module id or package name.

2.2. Configurations

2.2.1. hivemind.ApplicationDefaults Configuration

The ApplicationDefaults configuration is used to set default values for substitution symbols. Application defaults override contributions to hivemind.FactoryDefaults. The contribution format is the same FactoryDefaults:

```
default (symbol=... value=...)
```

2.2.2. hivemind.EagerLoad Configuration

The EagerLoad configuration allows services to be constructed when the Registry is first initialized. Normally, HiveMind goes to great lengths to ensure that services are only constructed when they are first needed. Eager loading is appropriate during development (to ensure that services are configured properly), and some services that are event driven may need to be instantiated early, so that they may begin receiving event notifications even before their first service method is invoked.

Care should be taken when using services with the pooled or threaded service models to invoke cleanup the thread immediately after creating the Registry.

Contributions are as follows:

```
load (service-id=...)
```

2.2.3. hivemind.FactoryDefaults Configuration

The FactoryDefaults configuration is used to set default values for substitution symbols. Contributions look like:

```
default (symbol=... value=...)
```

Values defined here can be overriden by making a contribution to hivemind.ApplicationDefaults.

2.2.4. hivemind.ObjectProviders Configuration

The ObjectProviders configuration drives the `object` translator. Contributions define an object *provider* in terms of a prefix (such as `service`) and a service that implements the `ObjectProvider` interface. .

The contribution format defines the name and class for each service model:

```
provider (prefix=... service-id=...)
```

Prefixes must be unique.

The following default prefixes are available:

Prefix	Descripton	Example
bean	The locator is a BeanFactory locator, consisting of the id of a BeanFactory service, a colon, and an optional initializer for the bean. **Note:** Provided by the HiveMind library.	bean:ValidatorFactory:string,required
configuration	The locator is the id of a configuration.	configuration:MyConfiguration
instance	The locator is a fully qualified class name, which must have a public no arguments contructor.	instance:com.example.MyObject
service	The locator is the id of a service.	service:MyService
service-property	The locator provides a service id and a property name (provided by that service), seperated with a colon.	service-property:MyService:activeRequest

2.2.5. hivemind.ServiceModels Configuration

The ServiceModels configuration defines the available service models. Service models control the lifecycle of services: when they are created and when they are destroyed (often tied to the current thread's activity).

The contribution format defines the name and class for each service model:

```
service-model (name=... class=...)
```

An instance of the specified class will be instantiated. The class must implement the ServiceModelFactory interface (which creates an instance of the actual service model for a particular service extension point).

Names of service models must be unique; it is not possible to override the built-in service model factories.

2.2.6. hivemind.SymbolSources Configuration

The SymbolSources configuration is used to define new SymbolSources (providers of values for substitution symbols).

Contributions are of the form:

```
source (name=... before=... after=... class=... service-id=...)
```

Sources are ordering based on the `name`, `before` and `after` elements. `before` and `after` may be comma-seperated lists of other sources, may be the simple value `*`, or may be omitted.

Only one of `class` and `service-id` attributes should be specified. The former is the complete name of a class (implementing the SymbolSource interface). The second is used to contribute a service (which must also implement the interface).

2.2.7. hivemind.Translators Configuration

The Translators configuration defines the translators that may be used with XML conversion rules.

The contribution format defines the name and class for each service model:

```
translator (name=... class=...)
```

An instance of the specified class will be instantiated. The class must implement the Translator interface. It should have a no-args and/or single String constructor.

Names of translators must be unique; it is not possible to override the existing service model translators. A single translator, `class` , is hard-coded into HiveMind, the others appear as ordinary contributions.

2.3. Ant Tasks

2.3.1. ConstructRegistry Ant Task

Reads some number of HiveMind module descriptors and assembles a single registry file from them. The output registry consists of a <registry> element which contains one <module> element for each module descriptor read. This registry is useful for generating documentation.

The registry XML is only updated if it does not exist, or if any of the module deployment descriptor is newer.

This task is implemented as org.apache.hivemind.ant.ConstructRegistry.

2.3.1.1. Parameters

Attribute	Description	Required
output	The file to write the registry to.	Yes

2.3.1.2. Parameters specified as nested elements

descriptors

A path-like structure, used to identify which HiveMind module descriptors (`hivemodule.sdl` and `hivemind.xml`) should be included.

Each path element should either be a module deployment descriptor, or be a JAR containing a deployment descriptor (in the `META-INF` folder).

2.3.1.3. Examples

Create `target/registry.xml` from all `hivemodule.xml` descriptors found inside the `src` directory.

```
<constructregistry output="target/registry.xml">
  <descriptors>
    <fileset dir="src">
      <include name="**/hivemodule.xml"/>
    </fileset>
  </descriptors>
</constructregistry>
```

2.3.2. ManifestClassPath Ant Task

Converts a classpath into a space-separated list of items used to set the `Manifest Class-Path` attribute.

This is highly useful when modules are packaged together inside an Enterprise Application Archive (EAR). Library modules may be deployed inside an EAR, but (in the current J2EE specs), there's no way for such modules to be added to the classpath in the deployment descriptor; instead, each JAR is expected to have a Manifest Class-Path attribute identifying the exactly list of JARs that should be in the classpath. This Task is used to generate that list.

This task is implemented as org.apache.hivemind.ant.ManifestClassPath.

2.3.2.1. Parameters

Attribute	Description	Required
property	The name of a property to set as a result of executing the task.	Yes
directory	If specified, then the directory attribute does two things: • It acts as a filter, limiting the results to just those elements that are within the directory • It strips off the directory as a prefix (plus the separator), creating results that are relative to the directory.	No

2.3.2.2. Parameters specified as nested elements

classpath

A path-like structure, used to identify what the classpath should be.

2.3.2.3. Examples

Generate a list of JARs inside the `${target}` directory as relative paths and use it to set the Class-Path manifest attribute.

```
<manifestclasspath directory="${target}" property="manifest.class.path">
  <classpath refid="build.class.path"/>
</manifestclasspath>

<jar . . .>
  <manifest>
    <attribute name="Class-Path" value="${manifest.class.path}"/>
    . . .
  </manifest>
</jar>
```

2.4. Reports

2.4.1.

2.4.2.

3. Module: hivemind.lib

3.1. Services

3.1.1. hivemind.lib.BeanFactoryBuilder Service

The BeanFactoryBuilder services is used to construct a BeanFactory instance. An BeanFactory will *vend out* instances of classes. A logical name is mapped to a particular Java class to be instantiated.

Client code can retrieve beans via the factory's `get()` method. Beans are retrieved using a *locator*, which consists of a name and an optional initializer seperated by commas. The initializer is provided to the bean via an alternate constructor that takes a single string parameter. Initializers are used, typically, to initialize properties of the bean, but the actual implementation is internal to the bean class.

3.1.1.1. Usage

The general usage is as follows:

```
invoke-factory (service-id=hivemind.lib.BeanFactoryBuilder)
{
  factory (vend-class=... configuration-id=... default-cacheable=...)
}
```

The `vend-class` attribute is the name of a class all vended objects must be assignable to (as a class or interface). This is used to validate contributed bean definitions. By default it is `java.lang.Object`.

The `configuration-id` is the id of the companion configuration (used to define object classes).

The optional `default-cacheable` attribute sets the default for whether instantiated beans should be cached for reuse. By default this is true, which is appropriate for most use cases where the vended objects are immutable.

3.1.1.2. Configuration

Each BeanFactory service must have a configuration, into which beans are contributed:

```
configuration-point (id=... schema-id=hivemind.lib.BeanFactoryContribution)
```

Contributions into the configuration are used to specify the bean classes to instantiate, as:

```
bean (name=... class=... cacheable=...)
```

`name` is a unique name used to reference an instance of the class.

`class` is the Java class to instantiate.

`cacheable` determines whether instances of the class are cacheable (that is, have immutable internal state and should be reused), or non-cacheable (presumably, because of mutable internal state).

3.1.2. hivemind.lib.DefaultImplementationBuilder Service

The DefaultImplementationBuilder service is used to create default implementations of interfaces. As described in the service interface JavaDoc, methods return null, 0 or false (depending on return type) and otherwise do nothing.

3.1.3. hivemind.lib.EJBProxyFactory Service

The EJBProxyFactory service is used to construct a HiveMind service that delegates to an EJB stateless session bean. The EJB's remote interface is the service interface. When the first service method is invoked, the fabricated proxy will perform a JNDI lookup (using the NameLookup service), and invokes `create()` on the returned home interface.

The single service instance will be shared by all threads.

The general usage is as follows:

```
invoke-factory (service-id=hivemind.lib.EJBProxy)
{
  parameters (home-interface=... jndi-name=... name-lookup-service=...)
}
```

The `home-interface` attribute is the complete class name for the home interface, and is required.

The `jndi-name` attribute is the name of the EJB's home interface, also required.

The `name-lookup-service-id` attribute is optional and rarely used; it is an alternate service implementing the NameLookup interface to be used for JNDI lookups.

3.1.4. hivemind.lib.NameLookup Service

The NameLookup service is a thin wrapper around JNDI lookup. It is used by the EJBProxyFactory service to locate EJBs.

The implementation makes use of three symbols (all of whose values default to null):

- java.naming.factory.initial
- java.naming.factory.url.pkgs
- java.naming.provider.url

By supplying overrides of these values, it is possible to configure how the NameLookup service generates the InitialContext used for performing the JNDI lookup.

3.1.5. hivemind.lib.PlaceholderFactory Service

The PlaceholderFactory service is a service implementation factory that uses the DefaultImplementationBuilder service to create placeholder implementations for services. Placeholders do nothing at all.

3.1.6. hivemind.lib.PipelineFactory Service

The PipelineFactory services is used to construct a *pipeline* consisting of a series of filters. The filters implement an interface related to the service interface.

Each method of the service interface has a corresponding method in the filter interface with an identical signature, except that an additional parameter, whose type matches the *service interface* has been added.

For example, a service interface for transforming a string:

```
package mypackage;

public interface StringTransformService
{
  public String transform(String inputValue);
}
```

The corresponding filter interface:

```
package mypackage;

public interface StringTransformFilter
{
  public String transform(String inputValue, StringTransformService service);
}
```

The service parameter may appear at any point in the parameter list, though the convention of listing it last is recommended.

The filters in a pipeline are chained together as follows:

Pipeline Calling Sequence

The bridge objects implement the *service interface* (and are created dynamically at runtime). The *terminator* at the end also implements the service interface. This can be an object or a service; if no terminator is specified, a default implementation is created and used. Only a single terminator is allowed.

A pipeline is always created in terms of a service and a configuration. The service defines the service interface and identifies a configuration. The configuration conforms to the `hivemind.lib.Pipeline` schema and is used to specify filters and the terminator. Filters may be ordered much like <interceptor>s, using `before` and `after` attributes. This allows different modules to contribute filters into the service's pipeline.

3.1.6.1. Usage

The general usage is as follows:

```
invoke-factory (service-id=hivemind.lib.PipelineFactory)
{
  create-pipeline (filter-interface=... configuration-id=... terminator-service-id=...)
}
```

The `filter-interface` attribute is the complete class name of the filter interface.

The `configuration-id` is the id of the companion configuration (used to define filters).

The optional `terminator-service-id` attribute is used to specify a terminator service directly (a terminator may also be contributed into the pipline configuration).

3.1.6.2. Configuration

Each pipeline service must have a configuration, into which filters are contributed:

```
configuration-point (id=... schema-id=hivemind.lib.Pipeline)
```

3.1.6.3. Contributions

Contributions into the configuration are used to specify the filters and the terminator. Filters and terminators can be specified as services or as objects.

filter

```
filter (service-id=... before=... after=...)
```

Contributes a filter as a service. The optional `before` and `after` attributes are lists of the ids of other filters in the pipeline, used to set the ordering of the filters. They may be comma-seperated lists of filter ids (or filter names), or simple `*` to indicate absolute positioning.

filter-object

```
filter-object (name=... object=... before=... after=...)
```

Contributes a filter as an instance of the provided class. The `name` attribute is required and will be qualified with the contributing module id. The `object` attribute defines the object that will act as a filter. `before` and `after` are optional, as with the <filter> element.

terminator

```
terminator (service-id=...)
```

Specifies the terminator service for the pipeline. Only a single terminator may be specified, and the terminator service provided in the factory parameters takes precendence over a terminator in the configuration.

terminator-object

```
terminator-object (class=...)
```

Specifies the termnator for the pipeline as an object (intead of as a service).

3.1.7. hivemind.lib.RemoteExceptionCoordinator Service

The RemoteExceptionCoordinator is used to propogate notifications of remote exceptions throughout the HiveMind repository. When any individual service encounters a remote exception, it notifies all listeners, who release all remote object proxies.

The service interface, RemoteExceptionCoordinator, allows objects that implement the RemoteExceptionListener interface to be registered for notification, and includes a method for firing notifications.

3.1.8. hivemind.lib.SpringLookupFactory Service

The SpringLookupFactory supports integration with the Spring framework, another open-source lightweight container. SpringLookupFactory is a service constructor that obtains a core service implementation from a Spring `BeanFactory` .

By default, the `BeanFactory` is obtained from the DefaultSpringBeanFactoryHolder. Part of your application startup code requires that you start a Spring instance and inform the DefaultSpringBeanFactoryHolder about it.

The SpringLookupFactory expects exactly *one* parameter element:

```
lookup-bean (name=... source-service-id=...)
```

The `name` attribute is the name of the bean to look for inside the Spring BeanFactory.

The optional `source-service-id` attribute allows an alternate service to be used to obtain the BeanFactory. The identified service must implement the SpringBeanFactorySource interface.

3.2. Reports

3.2.1.

3.2.2.

4. Example Code

4.1. Calculator

The calculator example demonstrates the most basic concepts of HiveMind; the difference between <create-instance> and <invoke-factory>, the fact that services are, by default, created only as needed, and the ability of hivemind.BuilderFactory to automatically wire services together. It also demonstrates the behavior of the hivemind.LoggingInterceptor.

After compiling the examples, you can use Ant to run them:

```
bash-2.05b$ ant run-calculator
Buildfile: build.xml

run-calculator:
     [java] Calculator [DEBUG] Creating SingletonProxy for service examples.Calculator
     [java] Inputs:   28.0 and 4.75
     [java] Calculator [DEBUG] Constructing core service implementation for service exa
     [java] Subtracter [DEBUG] Creating SingletonProxy for service examples.Subtracter
     [java] Calculator [DEBUG] Autowired service property subtracter to <SingletonProxy
     [java] Divider [DEBUG] Creating SingletonProxy for service examples.Divider
     [java] Calculator [DEBUG] Autowired service property divider to <SingletonProxy fo
     [java] Multiplier [DEBUG] Creating SingletonProxy for service examples.Multiplier
     [java] Calculator [DEBUG] Autowired service property multiplier to <SingletonProxy
     [java] Adder [DEBUG] Creating SingletonProxy for service examples.Adder
     [java] Calculator [DEBUG] Autowired service property adder to <SingletonProxy for
     [java] Calculator [DEBUG] Applying interceptor factory hivemind.LoggingInterceptor
     [java] Calculator [DEBUG] BEGIN add(28.0, 4.75)
     [java] Adder [DEBUG] Constructing core service implementation for service examples
     [java] Adder [DEBUG] Applying interceptor factory hivemind.LoggingInterceptor
     [java] Adder [DEBUG] BEGIN add(28.0, 4.75)
     [java] Adder [DEBUG] END add() [32.75]
     [java] Calculator [DEBUG] END add() [32.75]
     [java] Add:      32.75
     [java] Calculator [DEBUG] BEGIN subtract(28.0, 4.75)
     [java] Subtracter [DEBUG] Constructing core service implementation for service exa
     [java] Subtracter [DEBUG] Applying interceptor factory hivemind.LoggingInterceptor
     [java] Subtracter [DEBUG] BEGIN subtract(28.0, 4.75)
     [java] Subtracter [DEBUG] END subtract() [23.25]
     [java] Calculator [DEBUG] END subtract() [23.25]
     [java] Subtract: 23.25
     [java] Calculator [DEBUG] BEGIN multiply(28.0, 4.75)
     [java] Multiplier [DEBUG] Constructing core service implementation for service exa
     [java] Multiplier [DEBUG] Applying interceptor factory hivemind.LoggingInterceptor
     [java] Multiplier [DEBUG] BEGIN multiply(28.0, 4.75)
     [java] Multiplier [DEBUG] END multiply() [133.0]
     [java] Calculator [DEBUG] END multiply() [133.0]
     [java] Multiply: 133.0
     [java] Calculator [DEBUG] BEGIN divide(28.0, 4.75)
     [java] Divider [DEBUG] Constructing core service implementation for service exampl
     [java] Divider [DEBUG] Applying interceptor factory hivemind.LoggingInterceptor
     [java] Divider [DEBUG] BEGIN divide(28.0, 4.75)
     [java] Divider [DEBUG] END divide() [5.894736842105263]
     [java] Calculator [DEBUG] END divide() [5.894736842105263]
     [java] Divide:   5.894736842105263

BUILD SUCCESSFUL
Total time: 3 seconds
```

The logging configuration enables logging for the `hivemind` logger; that and the logging interceptors produces quite a bit of output. You can see that a *proxy* is created for services initially, and that the "core service implementation" for the service is created later ... the core service implementation consists of an instance of the service's POJO class, wrapped with any interceptors (the logging interceptor, in this case).

The Registry is built from the following module deployment descriptor:

```
module (id=examples version="1.0.0")
{
  service-point (id=Adder interface=org.apache.hivemind.examples.Adder)
  {
    create-instance (class=org.apache.hivemind.examples.impl.AdderImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Subtracter interface=org.apache.hivemind.examples.Subtracter)
  {
    create-instance (class=org.apache.hivemind.examples.impl.SubtracterImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Multiplier interface=org.apache.hivemind.examples.Multiplier)
  {
    create-instance (class=org.apache.hivemind.examples.impl.MultiplerImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Divider interface=org.apache.hivemind.examples.Divider)
  {
    create-instance (class=org.apache.hivemind.examples.impl.DividerImpl)
    interceptor (service-id=hivemind.LoggingInterceptor)
  }

  service-point (id=Calculator interface=org.apache.hivemind.examples.Calculator)
  {
    invoke-factory (service-id=hivemind.BuilderFactory)
    {
      construct (class=org.apache.hivemind.examples.impl.CalculatorImpl)

      // Note: services are autowired (as long as there's exactly one
      // service-point implementing the interface).
    }
    interceptor (service-id=hivemind.LoggingInterceptor)
  }
}
```

The service-point for the Calculator service is very simple ... as the comment indicates, the BuilderFactory is capable of locating the other services (Adder, Subtracter, etc.) by their *interface*, rather than requiring `set-service` elements to connect properites to services (using the target service's ids). These *properties* of the Calculator *implementation* are *autowired* to the matching services. Autowriring works only because just a single service within the *entire* Registry implements the specific interface. You would see errors if no service implemented the interface, or if more than one did.

4.2. Panorama Startup

Panorama is a disguised version of WebCT's **Vista** application. Vista is a truly massive web application, consisting of thousands of Java classes and JSPs and hundreds of EJBs. Vista is organized as a large number of somewhat interrelated *tools* with an underlying substrate of *services*. In fact, HiveMind was originally created to manage the complexity of Vista.

Note:

The reality is that Vista, a commercial project, has continued with an older version of HiveMind. Panorama is based on original code in Vista, but has been altered to take advantage of many features available in more recent versions of HiveMind. Keeping the names seperate keeps us honest about the differences between a product actually in production (Vista) versus an idealized version used for demonstration and tutorial purposes (Panorama).

With all these interrelated tools and services, the simple act of starting up the application was complex. Many tools and services have *startup operations*, things that need to occur when the application first starts up within the application server. For example, the help service reads and caches help text stored within the database. The mail service creates periodic jobs to peform database garbage collection of deleted mail items. All told, Vista had over 40 different tasks to perform at startup ... many with subtle dependencies (such as the mail tool needing the job scheduler service to be up and running).

The *legacy* version of Vista startup consisted of a WebLogic startup class that invoked a central stateless session EJB. The startup EJB was responsible for performing all 40+ startup tasks ... typically by invoking a public static method of a class related to the tool.

This was problematic for several reasons. It created a dependency on WebLogic to manage startup (really, a minor consideration, but one nonetheless). More importantly, it created an unnecessary binding between the startup EJB and all the other code in all the other tools. These unwanted dependencies created ripple effects throughout the code base that impacted refactored efforts, and caused deployment problems that complicated the build (requiring the duplication of many common classes inside the startup EJB's JAR, to resolve runtime classloader dependencies).

Note:

It's all about class loaders. The class loader that loaded the startup EJB didn't have visibility to the contents of the other EJB JARs deployed within the Vista EAR. To satisfy WebLogic's ejbc command (EJB JAR packaging tool), and to succesfully locate the classes at runtime, it was necessary to duplicate many classes from the other EJB JARs into the startup EJB JAR. With HiveMind, this issue goes away, since the module deployment descriptors store the class *name*, and the *servlet thread's context class loader* is used to resolve that name ... and *it* has visibility to all the classes in all the EJB JARs.

4.2.1. Enter HiveMind

HiveMind's ultimate purpose was to simplify all aspects of Vista development and create a simpler, faster, more robust development environment. The first step on this journey, a trial step, was to rationalize the startup process.

Each startup task would be given a unique id, a title and a set of *dependencies* (on other tasks). How the task actually operated was left quite abstract ... with careful support for supporting the existing legacy approach (public static methods). What would change would be how these tasks were executed.

The advantage of HiveMind is that each *module* can contribute as many or as few startup tasks as necessary into the Startup configuration point as needed. This allows the startup logic to be properly *encapsulated* in the module. The startup logic can be easily changed without affecting other modules, and without having to change any single contentious resource (such as the legacy approach's startup EJB).

4.2.2. Startup task schema

The schema for startup tasks contributions must support the explicit ordering of execution based on dependencies. With HiveMind, there's no telling in what order modules will be processed, and so no telling in what order contributions will appear within a configuration point ... so it is necessary to make ordering explicit by giving each task a unique id, and listing dependencies (the ids of tasks that must precede, or must follow, any single task).

Special consideration was given to supporting legacy startup code in the tools and services; code that stays in the form of a public static method. As HiveMind is adopted, these static methods will go away, and be replaced with either HiveMind services, or simple objects.

The schema definition (with desriptions removed, for compactness) follows:

```
  schema (id=Tasks)
{
  element (name=task)
  {
    attribute (name=title required=true)
    attribute (name=id required=true)
    attribute (name=before)
    attribute (name=after)
    attribute (name=executable required=true translator=object)

    conversion (class=com.panorama.startup.impl.Task)
  }

  element (name=static-task)
  {
    attribute (name=title required=true)
    attribute (name=id required=true)
    attribute (name=before)
```

```
      attribute (name=after)
      attribute (name=class translator=class required=true)
      attribute (name=method)

      rules
      {
        create-object (class=com.panorama.startup.impl.Task)
        invoke-parent (method=addElement)

        read-attribute (attribute=id property=id)
        read-attribute (attribute=title property=title)
        read-attribute (attribute=before property=before)
        read-attribute (attribute=after property=after)

        create-object (class=com.panorama.startup.impl.ExecuteStatic)
        invoke-parent (method=setExecutable)

        read-attribute (attribute=class property=targetClass)
        read-attribute (attribute=method property=methodName)
      }
    }
  }
```

Note:

For more details, see the HiveDoc for the Tasks schema.

This schema supports contributions in two formats. The first format allows an arbitrary object or service to be contributed:

```
  task (id=mail title=Mail executable=service:MailStartup)
```

The `executable` attribute is converted into an object or service; here the `service:` prefix indicates that the rest of the string, `MailStartup`, is a service id (other prefixes are defined by the hivemind.ObjectProviders configuration). If this task has dependencies, the `before` and `after` attributes can be specified as well.

To support legacy code, a second option, `static-task`, is provided:

```
  contribution (configuration-id=panorama.startup.Startup)
  {
    static-task (id=discussions title=Discussions after=mail class=com.panorama.discuss
  }
```

The `static-task` element duplicates the `id`, `title`, `before` and `after` attributes, but replaces `executable` with `class` (the name of the class containing the method) and `method` (the name of the method to invoke, defaulting to "init").

4.2.3. Startup Service

The schema just defines what contributions *look like* and how they are converted to objects; we need to define a Startup configuration point using the schema, and a Startup service that uses the configuration point.

```
  configuration-point (id=Startup schema-id=Tasks)

  service-point (id=Startup interface=java.lang.Runnable)
  {
    invoke-factory (service-id=hivemind.BuilderFactory)
    {
      construct (class=com.panorama.startup.impl.TaskExecutor)
      {
        set-configuration (property=tasks configuration-id=Startup)
      }
    }
  }

  contribution (configuration-id=hivemind.Startup)
  {
    service (service-id=Startup)
  }
```

The `hivemind.Startup` configuration point is used to ensure that the Panorama Startup service is executed when the Registry itself is constructed.

4.2.4. Implementation

All that remains is the implementations of the service and task classes.

4.2.4.1. Executable.java

```
package com.panorama.startup;

/**
 * Much like {@link java.lang.Runnable}, but allows the caller
 * to handle any exceptions thrown.
 *
 * @author Howard Lewis Ship
 */
public interface Executable
{
    public void execute() throws Exception;
}
```

The Executable interface is implemented by tasks, and by services or other objects that need to be executed. It `throws Exception` so that exception catching and reporting can be centralized inside the Startup service.

4.2.4.2. Task.java

```
package com.panorama.startup.impl;
```

```
import org.apache.hivemind.impl.BaseLocatable;

import com.panorama.startup.Executable;

/**
 * An operation that may be executed. A Task exists to wrap
 * an {@link com.panorama.startup.Executable} object with
 * a title and ordering information (id, after, before).
 *
 * @author Howard Lewis Ship
 */
public class Task extends BaseLocatable implements Executable
{
    private String _id;
    private String _title;
    private String _after;
    private String _before;
    private Executable _executable;

    public String getBefore()
    {
        return _before;
    }

    public String getId()
    {
        return _id;
    }

    public String getAfter()
    {
        return _after;
    }

    public String getTitle()
    {
        return _title;
    }

    public void setExecutable(Executable executable)
    {
        _executable = executable;
    }

    public void setBefore(String string)
    {
        _before = string;
    }

    public void setId(String string)
    {
        _id = string;
    }
    public void setAfter(String string)
    {
        _after = string;
    }

    public void setTitle(String string)
    {
        _title = string;
    }

    /**
     * Delegates to the {@link #setExecutable(Executable) executable} object.
     */
    public void execute() throws Exception
    {
        _executable.execute();
    }

}
```

The Task class is a wrapper around an Executable object; whether that's a service, some arbitrary object, or a StaticTask.

4.2.4.3. ExecuteStatic.java

```
package com.panorama.startup.impl;

import java.lang.reflect.Method;

import com.panorama.startup.Executable;

/**
 * Used to access the legacy startup code that is in the form
 * of a public static method (usually <code>init()</code>) on some
 * class.
 *
 * @author Howard Lewis Ship
 */
public class ExecuteStatic implements Executable
{
    private String _methodName = "init";
    private Class _targetClass;

    public void execute() throws Exception
    {
        Method m = _targetClass.getMethod(_methodName, null);

        m.invoke(null, null);
    }

    /**
     * Sets the name of the method to invoke; if not set, the default is <code>init</co
     * The target class must have a public static method with that name taking no
     * parameters.
     */
    public void setMethodName(String string)
    {
        _methodName = string;
    }

    /**
     * Sets the class to invoke the method on.
     */
    public void setTargetClass(Class targetClass)
    {
        _targetClass = targetClass;
    }
}
```

ExecuteStatic uses Java reflection to invoke a public static method of a particular class.

4.2.4.4. TaskExecutor.java

```
package com.panorama.startup.impl;

import java.util.Iterator;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.hivemind.ErrorHandler;
import org.apache.hivemind.Messages;
import org.apache.hivemind.order.Orderer;

/**
 * A service that executes a series of {@link com.panorama.startup.impl.Task}s. Tasks h
 * an ordering based on pre- and post-requisites.
 *
 * @author Howard Lewis Ship
 */
public class TaskExecutor implements Runnable
{
    private ErrorHandler _errorHandler;
    private Log _log;
    private List _tasks;
    private Messages _messages;

    /**
     * Orders the {@link #setTasks(List) tasks} into an execution order, and executes
     * each in turn.  Logs the elapsed time, number of tasks, and the number of failure
     */
    public void run()
    {
        long startTime = System.currentTimeMillis();

        Orderer orderer = new Orderer(_log, _errorHandler, task());

        Iterator i = _tasks.iterator();
        while (i.hasNext())
        {
            Task t = (Task) i.next();

            orderer.add(t, t.getId(), t.getAfter(), t.getBefore());
        }

        List orderedTasks = orderer.getOrderedObjects();

        int failures = 0;

        i = orderedTasks.iterator();
        while (i.hasNext())
        {
            Task t = (Task) i.next();

            if (!execute(t))
                failures++;
        }

        long elapsedTime = System.currentTimeMillis() - startTime;

        if (failures == 0)
            _log.info(success(orderedTasks.size(), elapsedTime));
        else
            _log.info(failure(failures, orderedTasks.size(), elapsedTime));
    }

    /**
     * Execute a single task.
     *
     * @return true on success, false on failure
     */
    private boolean execute(Task t)
    {
        _log.info(executingTask(t));

        try
        {
            t.execute();

            return true;
        }
        catch (Exception ex)
        {
            _errorHandler.error(_log, exceptionInTask(t, ex), t.getLocation(), ex);

            return false;
        }
    }

    private String task()
    {
        return _messages.getMessage("task");
    }

    private String executingTask(Task t)
    {
        return _messages.format("executing-task", t.getTitle());
    }

    private String exceptionInTask(Task t, Throwable cause)
    {
        return _messages.format("exception-in-task", t.getTitle(), cause);
    }

    private String success(int count, long elapsedTimeMillis)
    {
        return _messages.format("success", new Integer(count), new Long(elapsedTimeMill
    }

    private String failure(int failureCount, int totalCount, long elapsedTimeMillis)
    {
        return _messages.format(
            "failure",
            new Integer(failureCount),
            new Integer(totalCount),
            new Long(elapsedTimeMillis));
    }

    public void setErrorHandler(ErrorHandler handler)
    {
        _errorHandler = handler;
    }

    public void setLog(Log log)
    {
        _log = log;
    }

    public void setMessages(Messages messages)
    {
        _messages = messages;
    }

    public void setTasks(List list)
    {
        _tasks = list;
    }

}
```

This class is where it all comes together; it is the core service implementation for the `panorama.startup.Startup` service. It is constructed by the hivemind.BuilderFactory, which autowires the `errorHandler`, `log` and `messages` properties, as well as the `tasks` property (which is explicitly set in the module deployment descriptor).

Most of the `run()` method is concerned with ordering the contributed tasks into execution order and reporting the results.

4.2.5. Unit Testing

Unit testing in HiveMind is accomplished by *acting like the container*; that is, your code is responsible for instantiating the core service implementation and setting its properties. In many cases, you will set the properties to mock objects ... HiveMind uses EasyMock extensively, and provides a base class, `HiveMindTestCase`, that contains much support for creating Mock controls and objects.

4.2.5.1. TestTaskExcecutor.java

```
package com.panorama.startup.impl;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Locale;

import org.apache.commons.logging.Log;
import org.apache.hivemind.ApplicationRuntimeException;
import org.apache.hivemind.ErrorHandler;
import org.apache.hivemind.Messages;
import org.apache.hivemind.Resource;
import org.apache.hivemind.impl.MessagesImpl;
import org.apache.hivemind.test.ExceptionAwareArgumentsMatcher;
import org.apache.hivemind.test.HiveMindTestCase;
import org.apache.hivemind.test.RegexpArgumentsMatcher;
import org.apache.hivemind.util.FileResource;
import org.easymock.MockControl;

import com.panorama.startup.Executable;

/**
 * Tests for the {@link com.panorama.startup.impl.TaskExecutor} service.
 *
 * @author Howard Lewis Ship
 */
public class TestTaskExecutor extends HiveMindTestCase
{
    private static List _tokens = new ArrayList();

    protected void setUp()
    {
        _tokens.clear();
    }

    protected void tearDown()
    {
        _tokens.clear();
    }

    public static void addToken(String token)
    {
        _tokens.add(token);
    }

    public Messages getMessages()
    {
      . . .
    }

    public void testSuccess()
    {
        ExecutableFixture f1 = new ExecutableFixture("f1");

        Task t1 = new Task();

        t1.setExecutable(f1);
        t1.setId("first");
        t1.setAfter("second");
        t1.setTitle("Fixture #1");

        ExecutableFixture f2 = new ExecutableFixture("f2");

        Task t2 = new Task();
        t2.setExecutable(f2);
        t2.setId("second");
        t2.setTitle("Fixture #2");

        List tasks = new ArrayList();
        tasks.add(t1);
        tasks.add(t2);

        MockControl logControl = newControl(Log.class);
        Log log = (Log) logControl.getMock();

        TaskExecutor e = new TaskExecutor();

        ErrorHandler errorHandler = (ErrorHandler) newMock(ErrorHandler.class);

        e.setErrorHandler(errorHandler);
        e.setLog(log);
        e.setMessages(getMessages());
        e.setTasks(tasks);

        // Note the ordering; explicitly set, to check that ordering does
        // take place.
        log.info("Executing task Fixture #2.");
        log.info("Executing task Fixture #1.");
        log.info("Executed 2 tasks \\(in \\d+ milliseconds\\)\\.");
        logControl.setMatcher(new RegexpArgumentsMatcher());
        replayControls();

        e.run();

        assertListsEqual(new String[] { "f2", "f1" }, _tokens);

        verifyControls();
    }

}
```

In this listing (which is a paired down version of the real class), you can see how mock objects, including EasyMock objects, are used. The ExecutableFixture classes will invoke the `addToken()` method; the point is to provide, in the tasks List, those fixtures wrapped in Task objects and see that they are invoked in the correct order.

We create a Mock Log object, and check that the correct messages are logged in the correct order. Once we have set the expectations for all the EasyMock controls, we invoke `replayControls()` and continue with our test. The `verifyControls()` method ensures that all mock objects have had all expected methods invoked on them.

That's just *unit* testing; you always want to supplement that with *integration* testing ... to ensure, at the very least, that your schema is valid, the conversion rules work, and the contributions are correct. However, as the code coverage report shows, you can reach very high levels of code coverage (and code *confidence*) using unit tests.

4.3. Reports

4.3.1.

4.3.2.

4.3.3.

5. Other Resources

6. Complete Site

7. Related Projects