FeatureRequest

A place to drop some ideas for new features for HiveMind

Performance Counter Enhancement

Allow performance counter resetting from a jmx console for the jmx performance collector interceptor.

Multiple Implementations for one service-point

Currently you can only give one implementation for a service point. Wouldn't it make sense to specify via a *occurs* attribute how many implementation you allow.

<service-point id="my.service.point" occurs="*"/>

Fetch a list of service implementations from the Registry

You could fetch a List of implementations like to the contributions:

List services = registry.getServices("service-point-id")

Select a specific service implementation from the Registry

If we put a new attribute to the <implementation> tag, for example *hint* or *name*, we could event select a implementation from the Registry:

<implementation service-id="my.service.point" hint="variantA"/>

<implementation service-id="my.service.point" hint="variantB"/>

IService myService = (IService) registry.getServices("my.service.point", "variantA", IService.class);

Inlining of the configuration into the implementation tag

Currently you have to configure a service via the <configuration-point> and <contribute> tag. The big advantage of this mechanism is, that you can provide additional configurations to an existing service.

At least in my case most components doesn't take a list of configuration items, but one configuration item. So most of the time the *occurs* tag has the value 1.

For reading and manipulating the configuration I have to look at two places:

- at the <contribute> tag where i have to specify the configuration
- and at the <implementation> tag where I create the service via <invoke-factory> and I have to specify the configuration-id.

Wouldn't it be nicer to inline the content of the <contribution> tag into the <implementation> tag?

So instead of:

```
<configuration-point id="config" occurs="1"><schema>...</schema></configuration-point>

<contribution configuration-id="config"/>my configuration setting</contribution>

<implementation service-id="my.service">
  <invoke-factory service-id="hivemind.BuilderFactory">
        <construct class="my.Implementation">
                <configuration>config</configuration>
        </construct>
  </invoke-factory>
</implementation>
```

I would like to write:

```
<implementation service-id="my.service">
  <invoke-factory service-id="hivemind.BuilderFactory">
        <construct class="my.Implementation">
                <inline-configuration>my configuration setting</inline-configuration>
        </construct>
  </invoke-factory>
</implementation>
```

Putting the object to be configured object onto the stack

Currently the configuration contributions are put into a list via the <invoke-parent method="addElement"/> tag. If the first object on the stack for the inlined configuration wouldn't be the SchemaProcessor but the created but not yet initialized service, we could avoid the detour via the configuration list.

DieterBogdoll

Discussions:

Support Globally Unique Services

The following module definition

```
module (id=hivemind version="1.0.0")
{
  service-point (id=ThreadLocalStorage interface=org.apache.hivemind.service.ThreadLocalStorage)
  {
  }
}
```

requires the following Registry lookup

```
ThreadLocalStorage tls = (ThreadLocalStorage) r.getService("hivemind.ThreadLocalStorage", ThreadLocalStorage.
class);
```

Which is an often copied piece of code. Why not, given a slightly different module definition

```
module (id=org.apache.hivemind.service version="1.0.0")
{
  service-point (id=ThreadLocalStorage interface=org.apache.hivemind.service.ThreadLocalStorage)
  {
  }
}
```

then support a simplified lookup for services whose intent is to only have a single implementation

```
ThreadLocalStorage tls = (ThreadLocalStorage) r.getService(ThreadLocalStorage.class);
```

The internals of the above lookup could look like

```
public Object getService(Class serviceClass) {
  return getService(serviceClass.getName(), serviceClass);
}
```

This would prevent the proliferation of constants interfaces whose only existence is to provide a string value that is as unique as the service interface class name.

An example of such potential proliferation is the Hivemind class.

```
public final class HiveMind
{
    /**
     * The full id of the {@link org.apache.hivemind.service.ThreadEventNotifier}
     * service.
     */
    public static final String THREAD_EVENT_NOTIFIER_SERVICE = "hivemind.ThreadEventNotifier";
}
```

Alternatively, provide constant string values for all services exposed from the hivemind module so users don't have to read the SDL to figure out the service id. But this would be an onerous task and would need to be constantly synched with the SDL itself - hence the initial proposal of making the service id and service interface class name interchangeable for "default" usage.

Separate Interface from Implementation

In general, there is nothing really 'Hivemind' about Hivemind based Services. These Services simply follow a regular pattern for wiring and invocation and express this in metadata. Taking this one step further these Services could be developed and specified using some, external, specification and Hivemind could merely be the container.

It is in this spirit that I propose the following. All of these feature additions could be implement alongside current features and have surfaced from attempting to introduce the Hivemind system into a community of industry standard pundits.

Publish module.xsd

A published schema would allow unknown tools to be able help the developer author the requisite metadata.

Allow for META-INF/module.xml, META-INF/module.sdl

The descriptor files found throughout the industry have a file name that is indicative of the root element it contains. This seems natural and doesn't introduce the 'hive' semantic into what would otherwise be a regular .jar file of classes and interfaces.

Allow for multiple access points into the Registry

A good litmus test of any architecture is its ability to be viewed from multiple angles. Hivemind lives up to this test by exposing the Registry API and the <set-service/>/Java Bean property methods of obtaining Services. Why not allow (and advocate) other means such as JNDI? An interesting article on theserverside.com attests to JNDI's often overlooked capabilities (http://www.theserverside.com/news/thread.tss?thread_id=26932). A quick example of how this might look for Services and Configurations would be:

```
Context c = new InitialContext();
MyService ms = (MyService) c.lookup("service:" + MyService.class.getName());
MyConfiguration mc = (MyConfiguration) c.lookup("configuration:" + MyConfiguration.class.getName());
```

Deliver distinct products for API, SPI, and Reference Implementation

Registry as an API

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

Registr r = RegistryBuilder.constructDefaultRegistry();
```

also requires the entire Hivemind framework .jar file. It could be replaced with

```
import org.apache.registry.*;

RegistryFactory rf = RegistryFactory.newInstance();
Registry r = rf.getRegistry();
```

which would only require a org-apache-registry.jar file. In addition to this factory developers would need access to the SPI for manipulating modules so

```
import org.apache.registry.*;
import org.apache.registry.spi.*;

public class MyObjectFactory implements ServiceImplementationFactory {
...
```

Hibernate Support

See HibernateSupport

Retrieve configuration as a Map

Quite often an implementation takes a configuration of required and unique attributes. It would be nice to retrieve (or set by HiveMind) this as a Map rather than a List in the set-configuration.

Assign an Alias to a service

It can be useful to access a service under more than one name. For instance, two components may require a database connection obtained from a Thread Local service. In the context of an application it may be desirable for these two components to share the same connection. Thus the implementation of the service should be defined in one place, and the two service-ids are both assigned to the same implementation.

Configure without a XML for easy "containment"

– see http://www.jroller.com/page/fate/20050328#picocontainer_tagnuts_from_a_thousand

Shutdown in sequence, flexible naming for the "shutdown" method

Two suggestions:

- Shut non-threaded services down in the reverse order in which they were started.
- Configurable shutdown method name in the same way we have for initialization methods (i.e. something like shutdown-method="stop")