ConditionalContributionsProposal

Problem Description

This document reflects the state of HiveMind on April 14 2004, with the code between 1.0-alpha-3 and 1.0-alpha-4.

HiveMind has a powerful and somewhat ingenious approach to defining services that, notably, separates a service's *interface* from it's *implementation*. It's completely possible for one module to define a <service-point> and for an entirely different module to provide the implementation using a <implementation>. This decoupling of interface from implementation is very powerful; it helps make dependencies between modules more manageable, but more importantly, it allows you to *plug in* different implementations simply by controlling which modules (that is, which JARs) are on the classpath.

The latter opens up some interesting possibilities with regards to testing. By controlling the classpath during testing, you can have a module provide a for-testing-purposes implementation of a service. For example, an implementation of a service that only "pretended" to interact with some kind of back end system. In this way, you can more easily test the behavior of the other services that make use of the service.

That's the theory anyway. In practice, this can be a bit hairy! Manipulating the classpath during testing is possible but can get messy.

There are other use cases where a more dynamic approach to selecting the implementation for a particular service point is desirable. In some cases, it would be very nice to *choose* one of a number of implementation based on the runtime circumstances. Is a particular JDK available? Is a particular class available?

Here's a concrete example:

Say you want to make use of regular expressions. In JDK 1.3 and earlier, you might want to make use of a library such as Jakarta ORO. In JDK 1.4 and above, there's the java.util.regex package. It would be nice to define a unified interface for regular expressions as a service and have any code that uses regular expressions work through that service.

For deployment in JDK 1.3 or earlier, it would be necessary to include the ORO libraries on the classpath, and regular expressions would be handled by ORO.

For deployment in JDK 1.4 or later, the ORO libraries would not be necessary, and the implementation would be based on the java.util.regex package.

Proposed Solution

Change the <implementation> element to add a new `if` attribute. The value for the attribute is an expression in a specially conceived expression language.

This expression is evaluated as the registry is constructed (note: to be determined whether during specification parse or during registry construction).

Where the expression evaluates to `true`, the <implementation> will be applied to the <service-point>. Where the expression evaluates to `false`, the <implementation> will be ignored.

It will still be an error to not provide exactly **one** <implementation>, but this check will be applied *after* the list of possible <implementation>s has been filtered down via expression evaluation.

Default Implementation

When no expression is provided in an <implementation>, then the <implementation> is considered the *default* implementation; used only when no more selective implementation (that is, an <implementation> with an `if` attribute) has already filled the slot.

Expression Language

The expression language allows the expression in terms of JDK version, System properties, and the existence of particular classes on the classpath.

```
expression ::= term
               term "or" expression
               term "and" expression
               "not(" expression ")"
               "(" expression ")"
               function-call

function-call ::= function-name "(" literal ")"

function-name ::= class | property | jdk

literal ::=  <string literal>
```

Note the use of the tokens `and` and `or` rather than their Java language equivalents (&& and |). This is reasonable for attributes within an XML document. If the syntax used "&&" for logical and, then it would have to be written as && which would be *hideous*.

So, potential expressions are:

- `jdk(1.4)` – The execution environment is JDK 1.4 or higher
- `not(jdk(1.4)) and class(org.apache.oro.text.regex.Perl5Compiler)` – JDK 1.3 or earlier and class Perl5Compiler is on the classpath
- `property(unit-test-mode)` if -Dunit-test-mode=true provided on JDK command line

class function

Evaluates its argument as a fully-qualified class name and returns true if such a class is available on the classpath.

jdk function

Evaluates its argument as a JDK version number (a dotted sequence number) and evaluates it against the runtime environment. Returns true if the actual JDK is the same as, or a subsequent release to, the provided JDK version number. For example, `jdk(1.4)` would match against releases 1.4.1, 1.4.1 _01, 1.4.2, 1.5, 1.5.1, etc.

property function

Used to check a JDK system property as with `Boolean.getBoolean()`.

Further Notes

Perhaps this same concept should be extended to <contribution> as well. The discussion in the ServicePreloadProposal gives one example of where this would be useful.

Perhaps there should be more power to check for the existence of modules ... or even services within modules.

Commentary

LukeBlanshard, 14 April 2004:

- Do you really need the property function? Can't you just use the ${symbol} notation?
- Answering the "when this should be evaluated" issue: it obviously can't happen at parse time, because people will need to be able to use the ${symbol} notation within expressions.

HowardLewisShip:

So ${unit-test-mode}? I can see that ... though I'm nervous about the ambiguities of doing it later. It also means that consistency checks that currently occur when building the registry get deferred even later, to service construction time.

LukeBlanshard, 17 April 2004:

I was thinking more like '"${interface.module.ServiceName}" == "implementation.module.ImplementationName"'. Then you just arrange your symbol sources to look in standard places for these definitions, and you have maximum configurability.

ChristianEssl:

I think the solution is insufficient and cluttered: To a module a service defined by the module is either a service with module provided implementation(s) or a service demanded by the module (implementation is provided by another module). This combined with overriding/default-impl gives four (abstract) types of services: A provided service implementations(s) may either be overwriten (Type I) or final (Type II), a demanded service may either have a default implemantation(s) (Type III) or none (Type IV). Type I and Type III are actually the same, both provide different default implementations which can be overwritten. Current Hivemind (without this proposal) supports only Type II and IV with one implementation. The current proposal also supports a Type I and Type III but with the restriction to have only one default implementation. Ie it would be impossible to provide your own regex-impl if the regex-module already defines for JDK1.4 and JDK1.3 and you use JDK1.4.

A little change could solve that: The implementation tag should be used for overriding only and should be unconditional. The service however should be allowed to provide differend conditional implementations which are only taken if there is no implementation. (So implementation are allowed even when the service-definiton provides an implementation). If TYPE II (final) is seen as important - I don't think so - also an extra attribute should be added to the service tag. (An alternative - of course my favorite - would be to replace the invoke-factory with a script 😃