Open Service Interface Definition

Osid

Document Release: 2.0
OSID version 2.0

The package org.osid includes three objects as part of the O.K.I.™ strategy for application design, development, and evolution. A central premise of the strategy is the desirability of efficient implementation substitution. As an application is developed, it can be convenient to introduce implemented functionality incrementally. This is helpful for staged deployment, testing, evaluating alternate implementation performance or other characteristics, introducing feature upgrades, and so on.[1]

O.K.I.™ addresses this strategy with two mechanisms. First, the org.osid Package includes Open Service Implementation Defintions (OSIDs) that are groups of Application Programming Interfaces (APIs or simply Java interfaces) that address the data management and behavior of a service. For example, the Authentication OSID includes the definition of the objects, methods, and types appropriate to the task of authenticating agents in a learning or similar system. By publishing these OSIDs and then having institutions and vendors develop implementations of OSIDs and applications that rely on those OSIDs, implementations can be reused and applications can leverage best-of-breed implementations. Similarly, applications can be more easily ported across institutions. Second, osid includes the OsidLoader, OsidManager, and OsidOwner classes. These classes provide a level of indirection to traditional object life-cycle management.

Some background first.[2] Let's suppose we have an "`osid.x`" package that includes the "x" SID that contains two interfaces, "A" and "B" as follows:

```
interface A
{
    void doSomething();
}

interface B
{
    int getSomething();
}
```

Let's suppose further that we have an implementation of the "x" OSID in package "`impl1`" and that this package has classes with the same names as the interfaces they implement.

```
class A implements osid.x.A
```

[1] Refer to “Portability” for more information.
[2] Refer to “Hello, OKI” for a concrete example of these techniques for `org.osid` and `osid.shared.Id`.

```
{
    void doSomething()
    {
        ... some implementation code
    }
}

class B implements osid.x.B
{
    int getSomething()
    {
        ... some implementation code
        return i;
    }
}
```

Traditionally, applications instantiate concrete implementations of an interface-implementing object by using the special "`new`" operator. This operator takes a fully qualified class name. If our application needs to construct an "A" and a "B", the statements might look like this:

```
osid.x.A a = new impl1.A();
a.doSomething();
osid.x.B b = new impl1.B();
int j = b.getSomething();
```

Now suppose we decide to change to an "`impl2`"'s implementation of the "x" OSID. Statements might look like this:

```
osid.x.A a = new impl2.A();
a.doSomething();
osid.x.B b = new impl2.B();
int j = b.getSomething();
```

The disadvantage with this approach is that every reference to impl1 had to be changed to impl2 and the source code must be recompiled. This can be burdensome and error-prone as the volume of code that references implementations increases. This burden might be a barrier to using alternate implementations.

O.K.I.™ overcomes this disadvantage with OsidLoader and OsidManager. O.K.I.™ OSIDs each include an interface that implements OsidManager. The implementation handles OsidOwner and configuration information; more on this later. The implementation, by convention, provides a "create" method for each interface in the OSID (in some cases interfaces within the OSID have the "create" methods). The purpose of the "create" method is primarily to offer a level of indirection to the "new" operator. For those familiar with Design Patterns the OsidManager is a Factory. Secondarily, the "create" method may set some data, for example a unique object Id. Our osid.x OSID would be expanded as follows:

```
interface A
{
    void doSomething();
}

interface B
{
    int getSomething();
}
```

```
interface XManager
{
    A createA();
    B createB();
}
```

`OsidLoader` is used by an application to instantiate a specific manager. The first argument of the OsidLoader's getManager() method is the name of the OSID package name; the second argument is the name of the implementing package; the third argument is the context; and the fourth is the additionalConfiguration (more on these last two later). For example, for "`impl1`" as follows:

```
// setup
XManager xManager =
  org.osid.OsidLoader.getManager("osid.x.XManager","impl1",context,null);

// application
osid.x.A a = xManager.create.A();
a.doSomething();
osid.x.B b = xManager.create.B();
int j = b.getSomething();
```

If we now want to use "`impl2`", we have:

```
// setup
XManager xManager =
  org.osid.OsidLoader.getManager("osid.x.XManager","impl2",context,null);

// application
osid.x.A a = xManager.create.A();
a.doSomething();
osid.x.B b = xManager.create.B();
int j = b.getSomething();
```

Note the application code is identical! This is the power of using the indirection of OsidLoader and OsidManager, namely, only the OsidLoader call changes and all other application code does not. Since the second argument to OsidLoader is a String, it can be paramaterized and determined at run-time, for example from a properties file. This allows an integrator to change the implementing package without recompilation.

Now to the subject of `OsidOwner`. It is useful to be able to share and augment contextual information during an application session and to optionally persist that information across sessions. `OsidContext` is a class that provides a simple place to store and retrieve information. OSID implementations and applications can create Serializable[3] objects meaningful to themselves and add, remove, and get them from `OsidContext` instances. The `OsidLoader` sets the owner in an OSID Manager instance to whatever was passed into `OsidLoader`. This allows implementations and applications to all use the same owner and thereby all have access to a common data pool.

As a further convenience, `OsidLoader` will look for a.properties file with the name of the OSID Manager. If such a file resides with the OSID Manager, it is loaded. These properties can be augmented by setting the additionalConfiguration argument in the getManager() call. The properties can also be set by an application directly although this will completely override any properties file settings. Through this mechanism and `OsidContext`, implementations and applications can be customized and integrated.

[3] `java.io.Serializable` in the Java™ language binding of the implementation