Extension Code Separation in OCAP Applications

When authoring applications designed to be deployed on a variety of devices supporting different combinations of OCAP extensions (DVR, HomeNetworking, FrontPanel, etc), it is important to structure your app code in a way that will not cause class loading errors. There is no perfect solution to this problem because the Java Virtual Machine specification gives implementers the freedom to perform classloading as aggressively as they wish. However, it seems likely that most embedded VM implementations will try to be as lazy as possible with regard to classloading to avoid the speed and memory implications of loading classes that will never be used.

There is, however, one aspect of classloading that can not be avoided -- class file "verification". Any class that is loaded by the JVM must be verified as described in Section 4.9 of the JVM Specification (http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html). One step of the verification process is known as "data-flow analysis". This process checks the byte code for every method defined in the class for things like stack overflow /underflow, valid local variable assignments, and valid JVM instruction codes. For our purposes, we must focus on the "valid local variable assigment" task.

We will use the OCAP DVR extension for the purposes of this discussion. On devices that do not support DVR functionality, the OCAP DVR classfiles will most likely not be present in the classpath and any attempt by an application to classload a DVR class file will result in a NoClassDefFoundError being thrown. The OCAP specification indicates that application shall rely on the presence of the "ocap.api.option.dvr" Java system property to determine whether or not a given OCAP extension is present at runtime. Applications that use a common code base to deploy on both DVR and non-DVR devices should take care to structure the code so as to avoid incidental classloading of DVR classes during the data-flow analysis phase of class file verification. Take a look at this example:

```
interface A
{
    public void foo();
}

class B implements A
{
    public void foo() {}
}

class C implements A, org.ocap.shared.dvr.RecordedService
{
    public void foo() {}
    // RecordedService method implementations here...
}

class MyApp
{
    private A m_a;
    public void test()
    {
        A a = new B();             // B is classloaded during verification
                                   // of MyApp because types on each side
                                   // of the assignment are different, but B
                                   // itself is not verified until this code
                                   // is actually executed

        if (System.getProperty("ocap.api.option.dvr") != null)
        {
            A dvr_a = new C();     // C is classloaded during verification
        }                          // of MyApp because types on each side of
                                   // the assignment are different.  Fails to
                                   // load RecordedServiceDVR class

        B b = new B();             // B would not be classoaded during
                                   // verification of MyApp because types
                                   // on each side of the assignment are the same

        if (System.getProperty("ocap.api.option.dvr") != null)
        {
            m_a = new C();         // C would not be classloaded during verification
        }                          // of MyApp because only local variable
                                   // assigments are checked.
    }
}
```

All of the verification indicated above takes place during classloading of class MyApp which is a class that is used for both DVR and non-DVR devices. As you can see, depending on your use of assignments to local variables, you can accidentally trigger a classload of a DVR class even though the code will never actually get executed.