BeanMachine Technical Brief

August 18, 2011

ZedaSoft, Inc.
2310 Gravel Dr
Fort Worth TX 76118
817-616-1000
www.zedasoft.com

Introduction

The Container-Based Architecture (CBA) for Simulation is a family of object-oriented software frameworks that together implement ZedaSoft's simulation system design "A Container-based Architecture for Simulation of Entities in a Time Domain" (U.S. patent 7,516,052).

CBA is a general system for managing the update of arbitrary data over time. It makes no assumptions about the data being managed; instead, it provides an object assembly model, execution life-cycle management, network distribution, and other general services that can be used to model specific problem domains in a flexible way.

The object assembly step – as well as many other functions supported by CBA – often require that object graphs be loaded from or written to persistent storage. CBA uses XML for the persistent storage representation, and uses a component named BeanMachine to reconstitute object graphs from XML and distill existing object graphs in memory to XML. BeanMachine is used for nearly all XML I/O within CBA.

BeanMachine is a general object initialization and object graph assembly engine. It is able to load and distill object graphs of arbitrary complexity using any classes visible to it without requiring any changes to the participating classes.

One of the higher profile uses of BeanMachine within CBA is to load and distill container object graphs on behalf of the Fractal Containment® object assembly model and execution life-cycle management mechanism. See ZedaSoft's Fractal Containment® Technical Brief for more information.

Why "BeanMachine"?

CBA is implemented using Sun/Oracle's Java Development Platform. Somewhere along the way the people at Sun decided to continue the Java association with coffee and use the term "beans" for classes that were implemented using a certain set of patterns that made it possible to examine, create, and initialize them in a predictable way. The name "BeanMachine", though apparently blithe, seemed appropriate for an engine that could initialize and assemble such objects into an object graph.

Motivations

At the time we started designing the first version of CBA the common legacy approach to building applications was to hard-code and link a fixed set of components without a plug-in extension capability. Such applications were limited in what they could support. If you needed a different combination of components you had to round up the developers and go through another cycle of requirements development, code modification, testing, and deployment. This was always a time-consuming cycle.

The closest such an approach could come to flexible dynamic component selection and assembly was to base the component choices on a potentially large and clumsy set of flags that directed an application to use one class or another, or perform selected functions one way or another. This assumed dynamic class loading was even possible since not all languages support this capability. Even in the best case this approach was limited because

the total set of what you might want to support had to be built into the program, and it was difficult to configure multiple instances of a particular type of object to behave different ways in the same application.

Because of these limitations we decided that CBA should push application assembly as close as possible to the end user, providing an à la carte approach that would allow end users to select and assemble components in the way that best supported their simulation requirements. “Select and assemble” is the critical phrase; users pick the components they want to use, and put them together the way that best meets their requirements, per subsystem, per entity, per scenario.

Managing Class Availability: Software Component Manager

BeanMachine is a general object initialization and object graph assembly engine. It creates the graph by consuming an XML description that specifies the classes to use for each object. But how does it know what classes are available?

In CBA this is the job of a mechanism named Software Component Manager (SCM). SCM's job is to provide BeanMachine and the rest of the Java runtime with a “CLASSPATH”, which is a list of all possible persistent storage locations from which classes and any associated resources can be loaded.

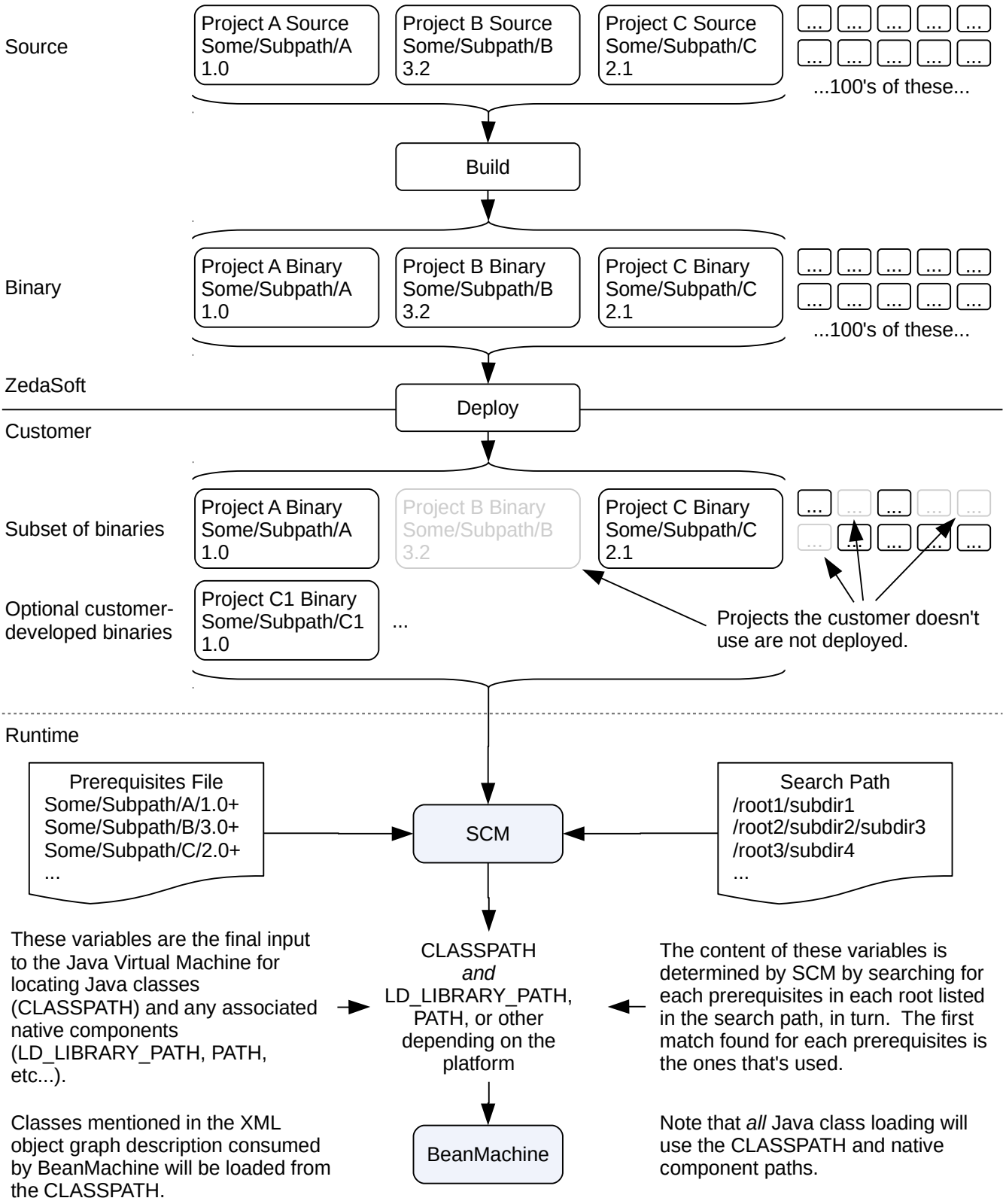
CBA is divided into projects, each of which is a deployment unit that contains a set of classes and other resources. Each project has a unique identifying subpath and version specifier. End-user simulations are composed using classes and resources from some subset of all possible projects. The required projects are declared in a prerequisites file that specifies the required project subpaths and a description of which project versions are acceptable.

SCM is provided with a search path that tells it the possible locations where each prerequisite project may possibly be resolved. The search path is nothing more than a list of possible roots against which each project subpath is applied to come up with a possible absolute persistent storage path for that project. To resolve a project SCM applies the project's subpath to each root in the search path in turn, and selects the first occurrence that meets the restrictions of the version specifier. This is done for each prerequisite project until all have been resolved. The end result is the CLASSPATH.

In general SCM is used only once at the time an application is started. Once the CLASSPATH has been established it remains the same for the lifetime of the application. BeanMachine bases all subsequent class loading on the CLASSPATH.

In addition to routine class location, SCM can be used to “slip in” different implementations for specific classes. This can for example be used to support field maintenance patches, changing classification levels, or replacing domestic use classes with approved-for-export counterparts.

SCM, BeanMachine, and Fractal Containment are of equal importance in CBA. SCM manages class availability, BeanMachine manages object initialization and cooperates with Fractal Containment to perform object graph assembly, and Fractal Containment manages object graph execution.


Source
Project A Source
Some/Subpath/A
1.0
Project B Source
Some/Subpath/B
3.2
Project C Source
Some/Subpath/C
2.1
...100's of these...
Build
Binary
Project A Binary
Some/Subpath/A
1.0
Project B Binary
Some/Subpath/B
3.2
Project C Binary
Some/Subpath/C
2.1
...100's of these...
ZedaSoft
Deploy
Customer
Subset of binaries
Project A Binary
Some/Subpath/A
1.0
Project B Binary
Some/Subpath/B
3.2
Project C Binary
Some/Subpath/C
2.1
Optional customer-developed binaries
Project C1 Binary
Some/Subpath/C1
1.0
...
Projects the customer doesn't use are not deployed.
Runtime
Prerequisites File
Some/Subpath/A/1.0+
Some/Subpath/B/3.0+
Some/Subpath/C/2.0+
...
SCM
Search Path
/root1/subdir1
/root2/subdir2/subdir3
/root3/subdir4
...
These variables are the final input to the Java Virtual Machine for locating Java classes (CLASSPATH) and any associated native components (LD_LIBRARY_PATH, PATH, etc...).
CLASSPATH
and
LD_LIBRARY_PATH, PATH, or other depending on the platform
The content of these variables is determined by SCM by searching for each prerequisites in each root listed in the search path, in turn. The first match found for each prerequisites is the ones that's used.
Classes mentioned in the XML object graph description consumed by BeanMachine will be loaded from the CLASSPATH.
BeanMachine
Note that all Java class loading will use the CLASSPATH and native component paths.

Describing the Object Graph: XML

BeanMachine initializes objects and weaves them into an object graph based on an XML description that specifies both the classes to instantiate and any initial property values they require. We refer to the XML as a "distilled" version of an object graph; it is an object graph expressed in the most refined form that still contains enough information to reconstitute the graph.

XML is a human-readable markup language much like HTML, except the rules about what constitutes valid XML can be much looser if you need them to be. Where HTML predefines the set of meaningful tags, XML lets *you* define the tags you need and decide how you want to enforce document validity.

On a very simple level you can think of an XML document as a markup document composed of "elements" and "attributes". These are syntactic concepts in XML. A simple element with some attributes might look like this:

```
<someObject
   intProperty="1"
   floatProperty="1.234"
/>
```

someObject is an element we defined for some TBD use.
intProperty and floatProperty are attributes. Attributes are typically reserved for simpler data, such as numeric values and character strings.

XML allows us to nest elements too:

```
<someObject
   intProperty="1"
   floatProperty="1.234"
>
   <someOtherObject
      stringProperty="Hello"
   />
</someObject>
```

We have inserted another element into our original element.

The fact that XML allows us to assemble hierarchical markup however we want using arbitrary element and attributes names makes it a suitable solution for describing or persisting a hierarchical graph of arbitrary objects. The XML document structure can directly mirror the structure of the object graph we need to describe or persist, and the elements and attributes can be used to name the property values of each object even if the values are complex.

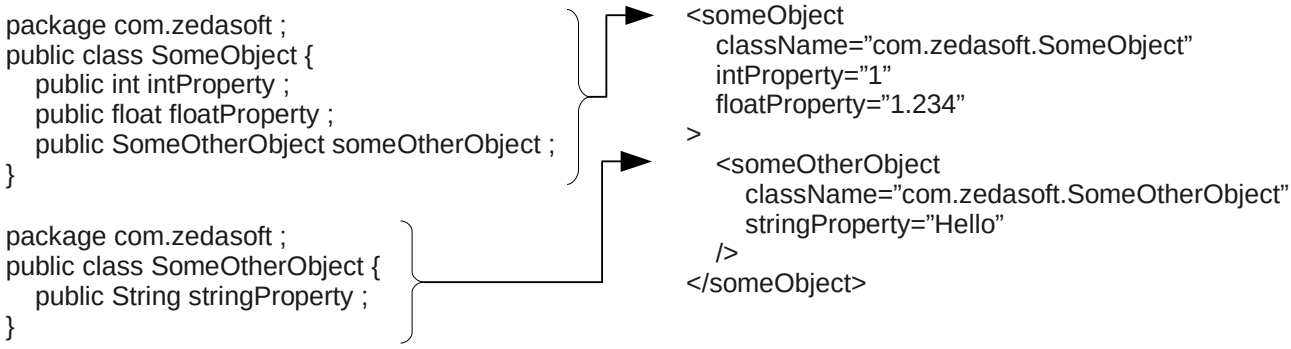
All that remains is knowing what element and attribute names to use and how they can be nested. In other words, what constitutes a valid BeanMachine-loadable XML document? By now you might be able to guess that since BeanMachine's goal is to create an object graph from XML, the XML element names, attribute and element values, and overall structure must be closely associated with the Java classes mentioned in the XML.

It turns out that this is exactly the case. BeanMachine-loadable XML is not constrained by a DTD or XML Schema. Instead, the structure of the XML directly mirrors the structure of the desired object graph, and the valid element and attributes names and values mirror the

corresponding property names and types in the instance being described in any particular element.

In a BeanMachine-loadable XML document there is a 1-to-1 relationship between each XML element and an instance of a Java class. You can use as many elements as you need, but each of them is describing exactly one instance. Note that this doesn't stop you from nesting elements because the nested elements are considered part of the description of their enclosing element. In other words, the general pattern is recursive. If the class being described has among its constituent parts instances of other classes, then we descend a level and the pattern starts over to describe the nested object.

Continuing and expanding our earlier example, we introduce two Java classes that define some public properties and show how they relate to the corresponding initialization XML:



Note that even though the two classes above are perfectly legal Java classes, having public fields like this is not necessarily the best practice, but it makes for a concise illustration in this document.

A "property" is either a declared instance variable (aka a "field") or a function defined on a class (aka a "method") that associates a name with a value. The value can be a simple primitive such as an integer or floating-point value, or a complex object such as SomeOtherObject in the example above.

Each element of our XML specifies what class should be instantiated, and the remainder of the attributes and elements specify property names and values that must be consistent with that class.

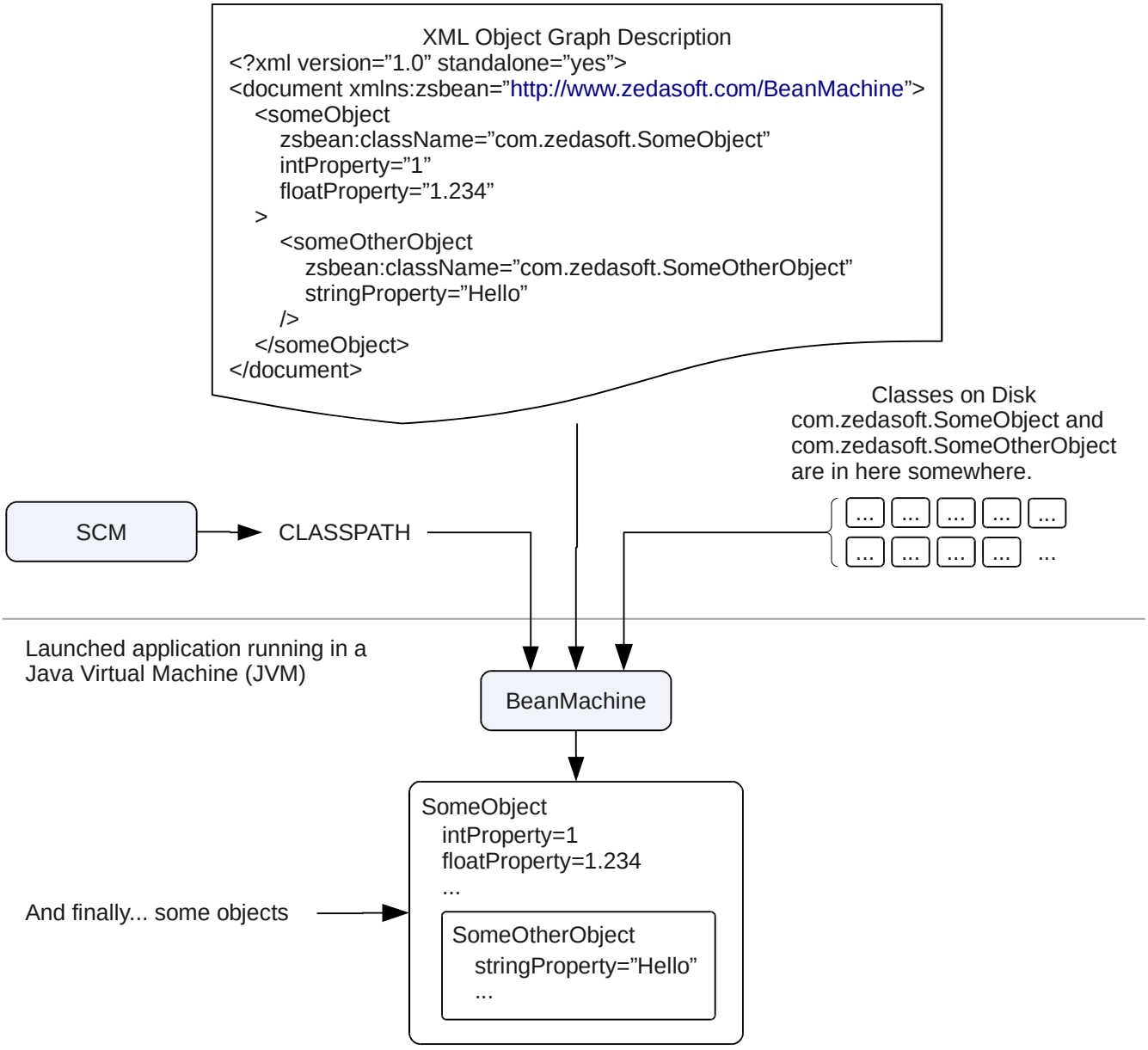
Object graphs can be simple or complex, and the corresponding XML description of the graph will be equally simple or complex. The fact that the graph description is only restricted by how the classes can be woven together is the most powerful aspect of this approach. It allows us to push object graph assembly all the way to the end users of the system where they can assemble objects limited only by what the classes say is permissible.

In summary, the structure of a BeanMachine-loadable XML document is as simple or as complex as the corresponding object graph. The names of the classes and the names of the properties in the classes in use define what can be used as XML element and attribute names. If the classes in use change, then any XML documents making use of those classes have to change accordingly.

Assembling the Object Graph: BeanMachine

Now that we have SCM describing where to find classes and XML describing which classes to instantiate and how to initialize the instances, we can let BeanMachine knit it all together into an in-memory object graph.

In this example we expand the XML from the previous section into a full BeanMachine-loadable document:



At this point you may be wondering why BeanMachine is such an important part of CBA. What is the value is being able to create a few simple objects with some fields initialized? After all, you could do that in code with a lot less trouble.

The answer lies in being able to describe an object graph of arbitrary complexity using classes of your choice including ones you developed yourself, constrained only by the class implementations.

BeanMachine In Action

When asked to reconstitute an object graph from an XML document BeanMachine will start scanning the document at the top element and create an instance of the specified class. Until another nested element is encountered this class will be responsible for deciding what attribute and element names are permissible. Let's call this the “active class”, and the associated instance the “active instance”.

The active class and active instance are pushed onto a stack, then document processing continues. Attributes are processed next. As BeanMachine encounters attribute names it asks the active class for the type of the property (field or method) whose name matches the attribute name. BeanMachine then converts the attribute value to be compatible with the property type and sets the value on the active instance. Each attribute is processed this way in turn.

Once the attributes are exhausted BeanMachine processes any nested elements. As with the attributes, the element names are assumed to be property names. BeanMachine consults the active class for the property type, converts the value, then sets the value on the active instance.

If an element is encountered that declares itself to be an instance of some other class, then an instance of that class is created and the instance and the class are pushed onto the stack. These become the new active instance and active class respectively. At this point processing is assumed to be for the new active class, and it now becomes responsible for deciding what attribute and element names are permissible.

When a closing tag is encountered for an element, the current active instance and active class are removed from the stack, and the active instance and class revert back to what they were before that element was encountered.

Processing continues in this way for the entire document. When the final closing tag is encountered the object graph is considered complete and is returned to whoever asked for it.

Getting the Most Out of BeanMachine

As discussed in an earlier section, the content of a BeanMachine-loadable XML document is driven directly by the classes being used. If you can't get away with something in code, then it isn't going to work in XML either.

To get the most out of BeanMachine your classes should be loosely coupled. In other words, you should go to great lengths to eliminate hard-coded class dependencies. If you do this, then you increase the chances that you will be able to interchange classes in an XML object graph description.

In Java the way to do this is through interfaces. Interfaces are declarations of API without the implementation. In C++ these would be called pure virtual classes. In Objective-C they would be called protocols. No matter what you call it, the goal is to define a named group of methods that any class is free to implement. Any object that implements an interface reasonably can be interchanged freely with any other object that implements the interface, no matter the actual classes of the objects.

Consider the following example:

```
package com.zedasoft ;
public interface SomeInterface {
    public void doSomething() ;
}
```

```
package com.zedasoft ;
public class Class1 implements SomeInterface {
    public void doSomething() {
        System.err.println( “class1” ) ;
    }
}
```

```
package com.zedasoft ;
public class Class2 implements SomeInterface {
    public void doSomething() {
        System.err.println( “class2” ) ;
    }
}
```

If class SomeObject declares a field in terms of the interface...

```
package com.zedasoft ;
public class SomeObject {
    public SomeInterface object ;
}
```

...then both of these initializations of a SomeObject instance are valid:

```
<someObject
    className=”com.zedasoft.SomeObject”
>
    <object className=”com.zedasoft.Class1”/>
</someObject>
```

```
<someObject
    className=”com.zedasoft.SomeObject”
>
    <object className=”com.zedasoft.Class2”/>
</someObject>
```

In one case we initialized someObject with an instance of Class1, and in the other case we used Class2. We can do this because both Class1 and Class2 implement SomeInterface, and SomeObject's “object” property is declared to be SomeInterface.

Summary

BeanMachine is a mechanism for loading object graphs from XML descriptions and distilling existing object graphs to XML. The XML is not constrained by a DTD or XML Schema and instead is restricted only by what the classes in use say is acceptable. Classes do not need to be modified in order to work with BeanMachine, and BeanMachine does not need advance knowledge of what classes might be used. This allows classes you developed yourself to be made available to BeanMachine and used as easily as classes provided by ZedaSoft.