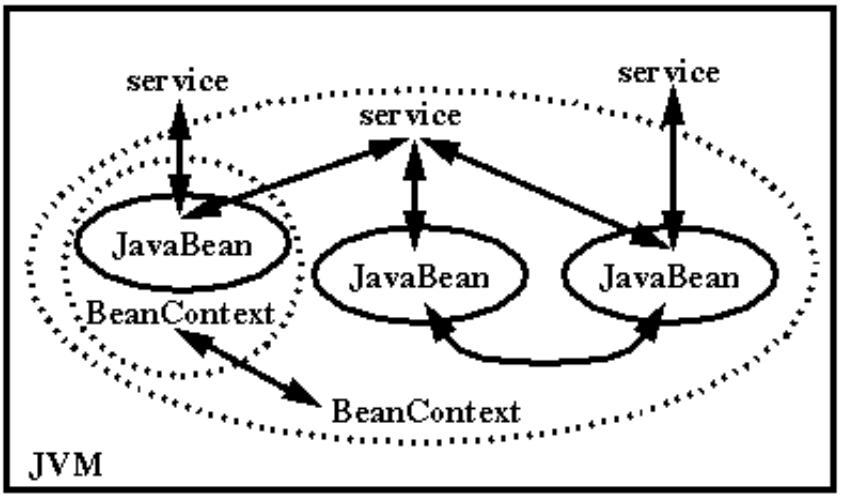
Extensible Runtime Containment and Services Protocol for JavaBeans Version 1.0

1.0 Introduction.

It is desirable to both provide a logical, traversable, hierarchy of JavaBeans, and further to provide a general mechanism whereby an object instantiating an arbitrary JavaBean can offer that JavaBean a variety of services, or interpose itself between the underlying system service and the JavaBean, in a conventional fashion.

The Container, or Embedding Context not only establishes the hierarchy or logical structure, but its also acts as a service provider that Components may interrogate in order to determine, and subsequently employ, the services provided by their Context.

- Introduce an abstraction for the environment, or context, in which a JavaBean logically functions during its lifecycle, that is a hierarchy or structure of JavaBeans.
- Enable the dynamic addition of arbitrary services to a JavaBean's environment.
- Provide a single service discovery mechanism through which JavaBeans may interrogate their environment in order both to ascertain the availability of particular services and to subsequently employ those services.
- Provide a simple mechanism to propagate an Environment to a JavaBean.
- Provide better support for JavaBeans that are also Applets.



2.0 API Specification

The hierarchal structure and general facilities of a *BeanContext* are provided for as follows:

```
public interface BeanContext extends java.beans.beancontext.BeanContextChild,
        java.util.Collection,
        java.beans.DesignMode,
        java.beans.Visibility {

    Object instantiateChild(String beanName) throws IOException,
ClassNotFoundException;

    public InputStream getResourceAsStream(String name, BeanContextChild
requestor);

    public java.net.URL getResource(String name, BeanContextChild requestor);

    void addBeanContextMembershipListener(BeanContextMembershipListener bcml);

    void removeBeanContextMembershipListener(BeanContextMembershipListener
bcml);

    public static final Object globalHierarchyLock;
}
```

```
public interface BeanContextMembershipListener extends java.util.Listener {
        void childrenAdded(BeanContextMembershipEvent   bcme);
        void childrenRemoved(BeanContextMembershipEvent bcme);
}
```

```
public abstract class BeanContextEvent extends  java.util.EventObject {
        public BeanContext getBeanContext();

        public synchronized void
                setPropagatedFrom(BeanContext bc);

        public synchronized BeanContext getPropagatedFrom();

        public synchronized boolean isPropagated()
}
```

```
public class BeanContextMembershipEvent extends BeanContextEvent {

        public BeanContextMembershipEvent(BeanContext bc,
                                          Object[]    deltas);

        public BeanContextMembershipEvent(BeanContext bc,
                                          Collection     deltas);

        public int size();

        public boolean contains(Object child);

        public Object[] toArray();

        public Iterator iterator();
}
```

One of the roles of the *BeanContext* is to introduce the notion of a hierarchical nesting or structure of *BeanContext* and JavaBean instances. In order to model this structure the *BeanContext* must expose API that defines the relationships in the structure or hierarchy.

The *BeanContext* exposes its substructure through a number of interface methods modeled by the *java.util.Collection* interface semantics

The *add()* method may be invoked in order to nest a new *Object*, *BeanContextChild*, or *BeanContext* within the target *BeanContext*. A conformant *add()* implementation is required to adhere to the following semantics:

- Synchronize on the *BeanContext.globalHierarchyLock.*

- Each child object shall appear only once in the set of children for a given *BeanContext*. If the instance is already a member of the *BeanContext* then the method shall return False.

- Each valid child shall be added to the set of children of a given source *BeanContext,* and thus shall appear in the set of children, obtained through either the *toArray(),* or *iterator()* methods, until such time as that child is deleted from the nesting *BeanContext* via an invocation of *remove()*, *removeAll()*, *retainAll()*, or *clear*()

- As the child is added to the set of nested children, and where that child implements the *java.beans.beancontext.BeanContextChild* interface (or *BeanContextProxy,* see later for details*)*, the *BeanContext* shall invoke the *setBeanContext*() method upon that child, with a reference to itself. Upon invocation, a child may, if it is for some reason unable or unprepared to function in that *BeanContext*, throw a *PropertyVetoException* to notify the nesting *BeanContext*. If the child throws such an exception the *BeanContext* shall revoke the addition of the child (and any other partial changes made to the state of the BeanContext as a side effect of this addition operation) to the set of nested children and throw an *IllegalStateException*.

- JavaBeans that implement the *java.beans.Visibility* interface shall be notified via the appropriate method, either *dontUseGui()* or *okToUseGui(),* of their current ability to render GUI as defined the policy of the *BeanContext*.

- If the newly added child implements *BeanContextChild*, the *BeanContext* shall register itself with the child on both its *VetoableChangeListener* and *PropertyChangeListener* interfaces to monitor, at least, that *BeanContextChild*'s "beanContext" property.

By doing so the *BeanContext* can monitor its child and can detect when such children are removed by a 3rd party (usually another *BeanContext*) invoking *setBeanContext*(). A *BeanContext* may veto such a change by a 3rd party if it determines that the child is not in a state to depart membership at that time.

- If the JavaBean(s) added implement *Listener* interfaces that the *BeanContext* is a source for, then the *BeanContext* may register the newly added objects via the appropriate *Listener* registration methods as a permissible side effect of nesting.

- If the JavaBean(s) added are Event Sources for Event that a particular *BeanContext* has

interest in the *BeanContext* may, as a side effect of adding the child, register *Listeners* on that child. The *BeanContext* should avoid using *Serializable Listeners* thus avoiding accidental serialization of unwanted structure when a child serializes itself.

- Once the *targetChild* has been successfully processed, the *BeanContext* shall fire a *java.beans.beancontext.BeanContextMembershipEvent*, containing a reference to the newly added *targetChild*, to the *childrenAdded*() method of all the *BeanContextMemebrshipListeners* currently registered.

- The method shall return true if successful.

- Synchronize with the *BeanContext.globalHierarchyLock.*

- If a particular child is not present in the set of children for the source *BeanContext,* the method shall return False.

- Remove the valid *targetChild* from the set of children for the source *BeanContext,* also removing that child from any other *Listener* interfaces that it was implicitly registered on, for that *BeanContext* as a side-effect of nesting.

Subsequently, if the *targetChild* implements the *java.beans.beancontext.BeanContextChild* interface (or *BeanContextProxy,* see later for deatils), the *BeanContext* shall invoke the *setBeanContext*() with a null[1] *BeanContext* value, in order to notify that child that it is no longer nested within the *BeanContext*.

If a particular *BeanContextChild* is in a state where it is not able to be un-nested from its nesting *BeanContext* it may throw a *PropertyVetoException*, upon receipt of this the *BeanContext* shall revoke the remove operation for this instance and throw *IllegalStateException*. To avoid infinite recursion, children are not permitted to repeatedly veto subsequent remove notifications. In practice, a child should attempt to resolve the condition (if temporary) that precludes it's removal from it's current nesting *BeanContext.*

- If the *targetChild* implements *java.beans.beancontext.BeanContextChild* then the *BeanContext* shall de-register itself from that child's *PropertyChangeListener* and *VetoableChangeListener* sources.

- If the *BeanContext* had previously registered the object(s) removed as *Listeners* on events sources implemented by the *BeanContext* as a side effect of nesting those objects, then the *BeanContext* shall de-register the newly removed object from the applicable source(s) via the appropriate *Listener* de-registration method(s)

- If the *BeanContext* had previously registered *Listener(s)* on the object(s) removed then the *BeanContext* shall remove those Listener(s) from those object(s).

- Once the *targetChild* has been removed from the set of children, the *BeanContext* shall fire a *java.beans.beancontext.BeanContextMembershipEvent,* containing a reference to the *targetChild* just removed, to the *childrenRemoved*() method of all the *BeanContextMembershipListeners* currently registered.

- Finally the method shall return the value true if successful.

BeanContext's are not required to implement either *addAll()*, *removeAll*(), *retainAll*() or *clear()* optional methods defined by *java.util.Collection API*, however if they do they must implement the semantics defined, per object, for both *add*() and *remove*() above. In the failure cases these methods shall revoke any partially applied changes to return the *BeanContext* to the state it was in prior to the failing composite operation being invoked, no *BeanContextEvents* shall be fired in the failure case as is consistent with the definition of *add()* and *remove()* above.

The *toArray(),* method shall return a copy of the current set of JavaBean or *BeanContext* instances nested within the target *BeanContext*, and the *iterator*() method shall supply a *java.util.Iterator* object over the same set of children.

The *size()* method returns the current number of children nested.

Note that all the *Collection* methods all require proper synchronization between each other by a given implementation in order to function correctly in a multi-threaded environment, that is, to ensure that any changes to the membership of the set of JavaBeans nested within a given *BeanContext* are applied atomically. All implementations are required to synchronized their implementations of these methods with the *BeanContext.globalHierarchyLock*.

The *instantiateChild()* method is a convenience method that may be invoked to instantiate a new JavaBean instance as a child of the target *BeanContext*. The implementation of the JavaBean is derived from the value of the *beanName* actual parameter, and is defined by the *java.beans.Beans.instantiate()* method.

The *BeanContextEvent* is the abstract root *EventObject* class for all *Events* pertaining to changes in state of a *BeanContext*'s defined semantics. This abstract root class defines the *BeanContext* that is the source of the notification, and also introduces a mechanism to allow the propagation of *BeanContextEvent* subclasses through a hierarchy of *BeanContexts*. The *setPropagatedFrom*() and *getPropagatedFrom*() methods allows a *BeanContext* to identify itself as the source of a propagated Event to the *BeanContext* to which it subsequently propagates the *Event* to. This is a general propagation mechanism and should be used with care as it has significant performance implications when propagated through large hierarchies.

Whenever a successful *add*(), *remove*(), *addAll*(), *retainAll*(), *removeAll*(), or *clear*() is invoked upon a particular *BeanContext* instance, a *BeanContextMembershipEvent* is fired describing the children effected by the operation.

The *BeanContext* defines two methods; *getResourceAsStream*() and *getResource*() which are analogous to those methods found on *java.lang.ClassLoader. BeanContextChild* instances nested within a *BeanContext* shall invoke the methods on their nesting *BeanContext* in preference for those on *ClassLoader*, to allow a *BeanContext* implementation to augment the semantics by interposing behavior between the child and the underlying *ClassLoader* semantics.

The service facilities of a *BeanContext* are provided as follows:

```
public interface BeanContextServices
          extends   BeanContext,BeanContextServicesListener {

        boolean addService(Class           serviceClass,
                           BeanContextServiceProvider service);

        boolean revokeService(Class           serviceClass,
                              BeanContextServiceProvider bcsp,
                              boolean         revokeNow
        );

        boolean hasService(Class serviceClass);

        Object getService(BeanContextChild      bcc,
                          Object                requestor.
                          Class                 serviceClass,
                          Object                serviceSelector,
                          BeanContextServicesRevokedListener sl
        ) throws TooManyListenersException;

        void releaseService(BeanContextChild bcc,
                            Object           requestor,
```

```
                            Object           service);

        Iterator getCurrentServiceClasses();

        public Iterator getCurrentServiceSelectors(Class sc);

        addBeanContextServicesListener(
                BeanContextServicesListener bcsl
        );

        removeBeanContextServicesListener(
                BeanContextServicesListener bcsl
     );
}
```

```
public interface BeanContextServiceProvider {
        Object getService(BeanContext bc,
                          Object      requestor,
                          Class       serviceCls,
                          Object      serviceSelector);

        void releaseService(BeanContext bc,
                            Object      requestor,
                            Object      service);

        Iterator getCurrentServiceSelectors(BeanContext bc,
                                            Class serviceCls);
}
```

The BeanContextServiceRevokedListener is defined as follows:

```
public interface BeanContextServiceRevokedListener
          extends java.util.EventListener {
        void serviceRevoked(
                BeanContextServiceRevokedEvent bcsre
        );
}
```

```
public interface BeanContextServicesListener
          extends BeanContextServiceRevokedListener {
        void serviceAvailable(
                BeanContextServiceAvailableEvent bcsae
        );
}
```

```
public class BeanContextServiceAvailableEvent
          extends BeanContextEvent {

        public BeanContextServiceAvailableEvent(
                        BeanContextServices        bcs,
                        Class                      sc
     );

        BeanContextServices getSourceAsBeanContextServices();

        public Class getServiceClass();

        public boolean isServiceClass(Class serviceClass);

        public Iterator getCurrentServiceSelectors();
}
```

```
public class BeanContextServiceRevokedEvent
          extends BeanContextEvent {
        public BeanContextServiceRevokedEvent(
                                BeanContextServices        bcs,
                                Class                      sc,
                            boolean                    invalidNow
        );
        public BeanContextServices
                        getSourceAsBeanContextServices();

        public Class getServiceClass();

        public boolean isServiceClass(Class service);

        public boolean isCurrentServiceInvalidNow();
}
```

```
public interface BeanContextServicesProviderBeanInfo
       extends   java.beans.BeanInfo {
    java.beans.BeanInfo[] getServicesBeanInfo();
}
```

A service, represented by a *Class* object, is typically a reference to either an interface, or to an implementation that is not publicly instantiable. This *Class* defines an interface protocol or contract between a *BeanContextServiceProvider*, the factory of the service, and an arbitrary object associated with a *BeanContextChild* that is currently nested within the *BeanContext* the service is registered with. Typically such protocols encapsulate some context specific or sensitive behavior that isolates a *BeanContextChild*'s implementation from such dependencies thus resulting in simpler implementations, greater interoperability and portability.

Once registered, and until revoked, the service is available via the *BeanContextServices getService*() method.

A *BeanContextChild* or any arbitrary object associated with a *BeanContextChild*, may obtain a reference to a currently registered service from its nesting *BeanContextServices* via an invocation of the *getService*() method. The *getService*() method specifies; the *BeanContextChild*, the associated *requestor*, the *Class* of the service requested, a service dependent parameter (known as a Service Selector), and a *BeanContextServicesRevokedListener* used to subsequently notify the requestor that the service class has been revoked by the *BeanContextServiceProvider*. The *Listener* is registered automatically with a unicast event source per requestor and service class and is automatically unregistered when a requestor relinquishes all references of a given service class, or as a side effect of the service being "forcibly revoked" by the providing *BeanContextServiceProvider*.

The reference to the *BeanContext* is intended to enable the *BeanContextServiceProvider* to distinguish service requests from multiple sources. A *BeanContextServiceProvider* is only permitted to retain a weak reference to any *BeanContext* so obtained.

The reference to the requestor is intended to permit the *BeanContextServiceProvider* to interrogate the state of the requestor in order to perform any customization or parameterization of the service, therefore this reference shall be treated as immutable by the *BeanContextServicesProvider*. Additionally the *BeanContextServiceProvider* is permitted to retain only weak and immutable reference to both the *requestor* and the *BeanContextChild* after returning from the *getService*() invocation.

In the case when a nested *BeanContextServices* is requested for a particular service that it has no *BeanContextServiceProvider* for, then the *BeanContextServices* may delegate the service requested to its own nesting *BeanContextServices* in order to be satisfied. Thus delegation requests can propagate from the leaf *BeanContextServices* to the root *BeanContextServices.*

If the service in question does not implement a finite set of apriori values for the set of valid Service Selectors it shall return null.

When *BeanContextChild* instances are removed from a particular *BeanContextServices* instance, they shall discard all references to any services they obtained from that *BeanContextServices* by appropriate invocations of *releaseService*(). If the un-nesting *BeanContextChild* is also a *BeanContextServices* instance, and if any of these service references have been exposed to the un-nesting *BeanContextServices*'s own members as a result of a delegated *getService*() request as defined above, the *BeanContextServiecs* shall fire a *BeanContextServiceRevokedEvent* to notify its nested children that the service(s) are "forcibly revoked". This immediate invalidation of current references to delegated services at un-nesting is to ensure that services that are dependent upon the structure of the hierarchy are not used by requestors after their location in the structure has changed.

A *BeanContextServiceProvider* may revoke a Service Class at any time after it has registered it with a *BeanContextServices* by invoking its *revokeService*() method. Once the *BeanContextServices* has fired a *BeanContextServiceRevokedEvent* notifying the currently registered *BeanContextServiceRevokedListeners* and the *BeanContextServicesListeners* that the service is now unavailable it shall no longer satisfy any new service requests for the revoked service until (if at all) that Service Class is re-registered. References obtained by *BeanContextChild* requestors to a service prior to its being revoked remain valid, and therefore the service shall remain valid to satisfy those extant references, until all references to that service are released, unless in exceptional circumstances the *BeanContextServiceProvider,* or *BeanContextServices*, when revoking the service, wants to immediately terminate service to all the current references. This immediate revocation is achieved by invoking the *BeanContextServices .revokeService*() method with an actual parameter value of *revokeNow* == true. Subsequent to immediate invalidation of current service references the service implementation may throw a service specific unchecked exception in response to any attempts to continue to use the revoked service by service requestors that have erroneously retained references to the service, ignoring the earlier immediate revocation notification.

A *BeanContextServicesProvider* may expose the *BeanInfo* for the Service Classes it provides implementations for by providing a *BeanInfo* class that implements *BeanContextServicesProviderBeanInfo.* Thus exposing an array of *BeanInfo*'s, one for each Service Class supported. Builder tools can, for example, use this infomation to provide application developers with a palette of Servlice Classes for inclusion in an application.

Since one of the primary roles of a *BeanContext* is to represent a logical nested structure of JavaBean component and *BeanContext* instance hierarchies, it is natural to expect that in many scenarios that hierarchy should be persistent, i.e that the *BeanContext* should participate in persistence mechanisms, in particular, either *java.io.Serializable* or *java.io.Externalizable (*If the latter the *BeanContext* is responsible for acting as the persistence container for the sub-graph of children, encoding and decoding the class information, and maintaining sub-graph equivalence after deserialization, basically the function(s) provide for serialization by *ObjectOutputStream* and *ObjectInputStream*).

As a result of the above requirement, persistent *BeanContextChil*d implementations are required to not persist any references to either their nesting *BeanContext*, or to any Delegates obtained via its nesting *BeanContextServices*.

Also note that since *BeanContext* implements *java.beans.beancontext.BeanContextChild* it shall obey the persistence requirements defined below for implementors of that interface.

Although not required, many *BeanContext*s may be associated within a presentation hierarchy of *java.awt.Container*s and *java.awt.Component*s. A *Container* cannot implement *BeanContext* directly[2] but may be associated with one by implementing the *BeanContextProxy* interface described herein.

```
public interface BeanContextProxy {
    BeanContext getBeanContext();
}
```

This also permits multiple distinct objects (e.g: *Containers*) to share a single *BeanContext*. [Note though that in this case a shared *BeanContext* shall not implement *BeanContextContainerProxy* since that is a peer-to-peer relationship between a single *BeanContext* and the *Container* implementing that interface]

No class may implement both the *BeanContext* (or *BeanContextChild*) and the *BeanContextProxy* interfaces, they are mutually exclusive.

In such cases it is possible to add, or remove, elements from either the *BeanContext,* via it's *Collection* API's, or the *BeanContextProxy* implementor using it's own collection-like API's (e.g: *public boolean java.awt.Container.add(Component)*). It is implementation dependent whether or not objects added or removed from either the *BeanContext*'s *Collection*, or the *BeanContextProxy* implementor's collection are also added or removed from the corresponding object's collection (i.e: should a *Container.add()* also infer a *BeanContext.add()* and vica-versa?). In such situations both participants (the implementor of *BeanContextProxy* and the *BeanContext* itself) are required to; 1) implement the same add/remove semantics as the other (i.e: if *x.add(o)* has a side effect of *x.getBeanContext().add(o)* then *x.getBeanContext().add(o)* should also a have side effect of *x.add(o)*), and 2) before adding/removing an object to/from the other participants collection, it should test (synchronized) if that object is/is not a member of the other participants collection before proceeding with the operation in question (this is to avoid infinite recursion between collection operations on both participants) (i.e: *x.add(o)* should not invoke *x.getBeanContext().add(o)* if *x.getBeanContext().contains(o)* is true and vica-versa).

The following interface is defined to allow a *BeanContext* to expose a reference to an associated *Container* to enable it's *BeanContextChild* members to add, or remove, their associated *Component* objects to/from that *Container* or to inspect some state on the *Container*.

```
public interface BeanContextContainerProxy {
        Container getContainer()
}
```

- If the associated *Component* was added to the associated *Container* via a *Container* API, then the nesting of the *BeanContextChild* in the *BeanContext* is a side effect of that and no further action is required.

- If the *Component* and *Container* are not nested then the nesting *BeanContext* may as a side effect cause the *Component* associated with the *BeanContextChild* to be added to it's associated *Container*.

OR

- If the *Component* and *Container* are not nested then the *BeanContextChild* being nested may as a side effect may cause it's *Component* to be associated with the *Container* associated with the nesting *BeanContext*.

The *BeanContextChild* is responsible for initially causing itself to eligible to be displayed via an invocation of *show*() [note that the *BeanContextChild* may also subsequently repeatedly *hide()* and *show()* itself].

Once a *BeanContextChild* has been un-nested from it's *BeanContext*, it's associated *Component* (if any) shall be removed from that *BeanContext*'s *Container* as a side effect of the removal operation, this is the responsibility of the *BeanContext* (typically if the *BeanContextChild* has been moved to another *BeanContext* with an associated *Container* via an invocation of it's *setBeanContext()* method, the *Component* will already have been re-parented as a side effect of that operation by the time the original *BeanContext* is notified of the change via a *PropertyChangeEvent* from the child, however the check should be made and the *Component* removed if it has not already occurred).

The value returned from the *getContainer()* method is constant for the lifetime of the implementing *BeanContext,* that is the relationship between a *BeanContext* and a *Container* is static for the lifetime of both participants.

```
public interface BeanContextChildComponentProxy {
    Component getComponent();
}
```

The value returned from the *getComponent()* method is a constant for the lifetime of that *BeanContextChild*.

If a class implements both *BeanContextChildComponentProxy* and *BeanContextContainerProxy* then the object returned by both *getComponent()* and *getContainer()* shall be the same object.

Simple JavaBeans that do not require any support or knowledge of their environment shall continue to function as they do today. However both JavaBeans that wish to utilize their containing *BeanContext*, and *BeanContext*s that may be nested, require to implement a mechanism that enables the propagation of the reference to the enclosing *BeanContext* through to cognizant JavaBeans and nested *BeanContext*s, the interface proposed is:

```
public interface java.beans.beancontext.BeanContextChild {
        void        setBeanContext(BeanContext bc)
                                throws PropertyVetoException;

        BeanContext getBeanContext();

        void addPropertyChangeListener
                (String name, PropertyChangeListener pcl);

     void removePropertyChangeListener
                (String name, PropertyChangeListener pcl);

        void addVetoableChangeListener
                (String name, VetoableChangeListener pcl);

     void removeVetoableChangeListener
                (String name, VetoableChangeListener pcl);

}
```

A *BeanContextChild* object may throw a *PropertyVetoException*, to notify the nesting *BeanContext* that it is unable to function/be nested within that particular *BeanContext*. Such a veto shall be interpreted by a *BeanContext* as an indication that the *BeanContextChild* has determined that it is unable to function in that particular *BeanContext* and is final.

Note that classes that implement this interface, also act as an Event Source for (sub)interface(s) of *java.beans.PropertyChangeListener*, and are required to update their state accordingly and subsequently fire the appropriate *java.beans.PropertyChangeEvent* with *propertyName* = "beanContext", *oldValue* = the reference to the previous nesting *BeanContext*, and *newValue* = the reference to the new nesting *BeanContext,* to notify any Listeners that its nesting *BeanContext* has changed value.

2.2.1 Important Persistence considerations

In order to ensure that the act of making such an instance persistent does not erroneously persist objects from the instances nesting environment, such instances shall be required to define such fields, or instance variables as either transient, or to implement custom persistence methods that avoid persisting such state.

3.0 Overloading java.beans.instantiate() static method

```
public static Object instantiate(ClassLoader cl,
                                 String        beanName,
                                 BeanContext beanContext);
```

4.0 Providing better support for Beans that are also Applets

Unfortunately this does not provide sufficient support in order to allow most Applets to be fully functional, since the *AppletContext* and *AppletStub* created by *java.beans.instantiate()*, are no-ops. This is a direct consequence of the lack of sufficient specification of how to construct *AppletContext* and *AppletStub* implementations in the existing *Applet* API's. Furthermore, even if such specifications existed we would require an API that propagated a number of *Applet* attributes such as its Codebase, Parameters, *AppletContext*, and Documentbase into

java.beans.instantiate() in order for it to subsequently instantiate the appropriately initialized objects.

```
public static Object
                        instantiate(ClassLoader
cl,
                                                String                beanName,
                                                BeanContext           bCtxt,
                                                AppletInitializer

                ai
                        );

public interface AppletInitializer {
        void initialize(Applet newApplet, BeanContext bCtxt);
        void activate(Applet newApplet);
}
```

Compliant implementations of *AppletInitializer.initialize()* shall:

- Associate the newly instantiated *Applet* with the appropriate *AppletContext*.
- Instantiate an *AppletStub*() and associate that *AppletStub* with the *Applet* via
- an invocation of *setStub*().
- If *BeanContext* parameter is null, then it shall associate the *Applet* with its appropriate *Container* by adding that *Applet* to its *Container* via an invocation of *add*(). If the *BeanContext* parameter is non-null, then it is the responsibility of the *BeanContext* to associate the *Applet* with its *Container* during the subsequent invocation of its *addChildren*() method.

Note that if the newly instantiated JavaBean is not an instance of *Applet*, then the *AppletInitializer* interface is ignored.

5.1 BeanContexts that support InfoBus.

A *BeanContext* that exposes an *InfoBus* to its nested *BeanContextChild* shall do so by exposing a service via the *hasService*() and *getService*() methods of type *javax.infobus.InfoBus.*

The Infobus 1.2 specification shall define a convenience mechanism provided by the *InfoBus* class to simplify the discovery mechanism for *BeanContextChild* instances nested within a particular instance of *BeanContextServices*.

A *BeanContext* that wishes to expose printing facilities to its descendants may delegate a reference of (sub)type *java.awt.PrintJob*.

5.3 BeanContext Design/Runtime mode support.

In the first version of the specification, the “mode” or state, that is “design”-time or “run”-time was a JVM global attribute. This is insufficient since, for example, in an Application Builder environment, there may be JavaBeans that function, in “run”-mode, as part of the Application Builder environment itself, as well as the JavaBeans that function, in “design”-mode, under construction by the developer using the Application Builder to compose an application.

The *BeanContext* abstraction, as a “Container” or “Context” for one or more JavaBeans provides appropriate mechanism to better scope this “mode”.

```
public interface java.beans.DesignMode {
        void    setDesignTime(boolean isDesignTime);
        boolean isDesignTime();
}
```

Note that it is illegal for instances of *BeanContextChild* to call *setDesignTime()* on instances of *BeanContext* that they are nested within.

JavaBeans with associated presentation, or GUI, may be instantiated in environments where the ability to present that GUI is either not physically possible (when the hardware is not present), or is not appropriate under the current conditions (running in a server context instead of a client).

*BeanContext*s that wish to enforce a particular policy regarding the ability of their children to present GUI, shall use the *java.beans.Visibility* interface to control their children.

*BeanContext*s may have a locale associated with them, in order to associate and propagate this important attribute across the JavaBeans nested therein.

Setting and getting the value of the *Locale* on the *BeanContext* is implementation dependent.

In order to ease the implementation of this relatively complex protocol a “helper” classes are provided; *java.beans.beancontext.BeanContextChildSupport, java.beans.beancontext.BeanContextSupport*, and *java.beans.beancontext.BeanContextServicesSupport*. These classes are designed to either be subclassed, or delegated implicitly by another object, and provides fully compliant (extensible) implementations of the protocols embodied herein.

2Unfortunately because of method name collisions between Component and Collection a Component can- not implement BeanContext or Collection directly and must model the capability with a “HasA” rather than an “IsA” relationship.

4Note: Since simple JavaBeans have no knowledge of a BeanContext, it is not advisable to introduce such instances into the hierarchy since there is no mechanism for these simple JavaBeans to remove them- selves from the hierarchy and thus subsequently be garbage collected.