Extensible Runtime Containment and Server Protocol for JavaBeans Version 1.0

Laurence Cable

Extensible Runtime Containment and Services Protocol for JavaBeans Version 1.0

Laurence Cable.

Send comments to java-beans@java.sun.com.

1.0 Introduction.

Currently the JavaBeans specification (Version 1.0) contains neither conventions describing a hierarchy or logical structure of JavaBeans, nor conventions for those JavaBeans to rendezvous with, or obtain arbitrary services or facilities from, the execution environment within which the JavaBean was instantiated.

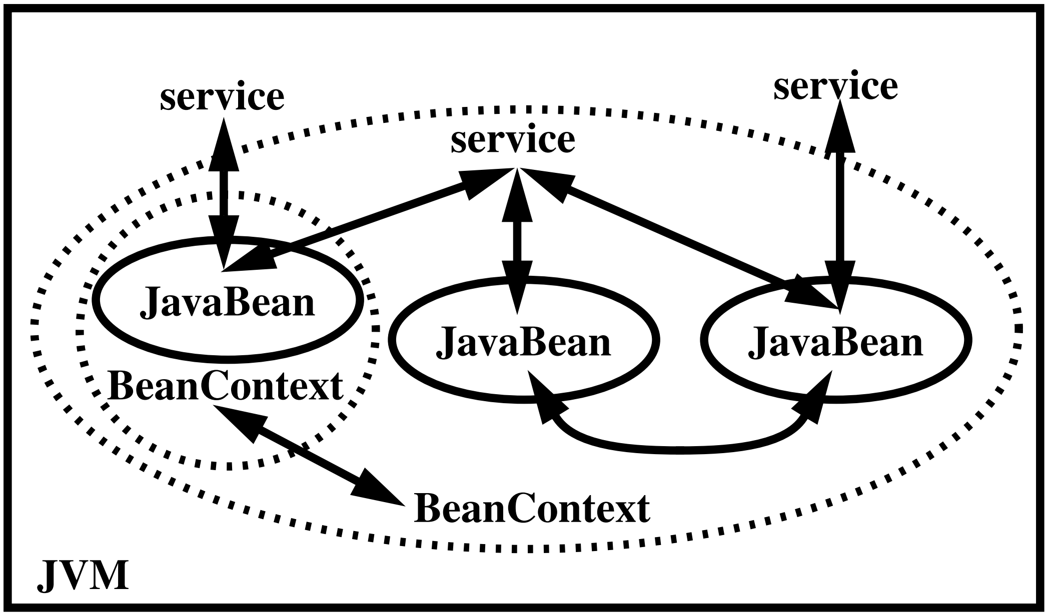
It is desirable to both provide a logical, traversable, hierarchy of JavaBeans, and further to provide a general mechanism whereby an object instantiating an arbitrary JavaBean can offer that JavaBean a variety of services, or interpose itself between the underlying system service and the JavaBean, in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

In other component models there exists the concept of a relationship between a Component and its environment, or Container, wherein a newly instantiated Component is provided with a reference to its Container or Embedding Context.

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.

This proposal defines such a protocol that supports extensible mechanisms that:

- 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

2.1 interface java.beans.beancontext.BeanContext

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

```
public interfacejava.beans.beancontext.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;
}
```

Notifications of changes in the membership of a *BeanContext* are modeled as follows:

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

The base class of all *BeanContext* related Events is defined by:

```
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()
}
```

The *BeanContextMembershipEvent* is defined as:

```
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();
}
```

2.1.1 The BeanContext as a participant in nested structure

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 superstructure through implementation of the *java.beans.beancontext.BeanContextChild* interface (as described later). This interface allows the discovery and manipulation of a *BeanContext*'s nesting *BeanContext*, and thus introduces a facility to create a hierarchy of *BeanContexts*.

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

BeanContexts are required to implement all the mandatory *Collection* API's, with the following particular semantics for *add*() and *remove*():.

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.

The *remove()* method may be invoked in order to remove an existing child JavaBean or *BeanContext* from within the target *BeanContext*. A conformant *remove()* implementation is required to adhere to the following semantics:

- 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 *IllegalState-*

Exception. 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.

Note that the lifetime of any child of a nesting *BeanContext*, is at least for the duration of that child's containment within a given *BeanContext*. For simple JavaBeans that are not aware of their containment within a *BeanContext*, this usually implies that the JavaBean shall exist for at least the lifetime of the nesting *BeanContext*.

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.

BeanContextMembershipListeners may be added and removed via invocations of *addBeanContextMembershipListener*() and *removeBeanContextMembershipListener*().

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 *contains()* method shall return `true` if the object specified is currently a child of the *BeanContext*.

1. Note, if the *remove*() was invoked as a result of the *BeanContext* receiving an unexpected *PropertyChangeEvent* notification as a result of a 3rd party invoking *setBeanContext*() then the remove implementation shall not invoke *setBeanContext*(`null`) on that child as part of the *remove*() semantics, since to do so would overwrite the value previously set by the 3rd party.

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

The *isEmpty*() method returns true iff the *BeanContext* has no children.

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*.

In some situations, *add()* and *remove()* (or a variant thereof) operations may occur nested, that is multiple occurrences may appear on the stack of the calling *Thread* simultaneously, e.g: when *BeanContextChild,* A, is added (or removed), it's *setBeanContext()* method also adds (or removes) another *BeanContextChild,* B. A particular *BeanContext* implementation may choose to fire either two *BeanContextMembershipListener* notifications, one for each *add()*/*remove()* operation of B then A (in this order since B is successfully added before A), or coalesce these into a single notification containing both A, and B. Note that should A be unable to be added or removed for any reason it shall not perform, or undo, any add or remove operations upon B as a side-effect, prior to throwing a *PropertyVetoException* to indicate this condition, that is, it must avoid or undo any side-effect membership changes prior to rejecting any changes to its own membership status.

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.

Typically, this shall be implemented by calling the appropriate *java.beans.Beans.instantiate()* method, using the *ClassLoader* of the target *BeanContext*. However a particular *BeanContext* implementation may interpose side-effects on the instantiate operation in their implementation of this 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.

The *BeanContextMembershipEvent* describes changes that occur in the membership of a particular *BeanContext* instance. This event encapsulates the list of children either added to, or removed from, the membership of a particular *BeanContext* instance, i.e the delta in the membership set.

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.

2.1.2 Resources.

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.

2.1.3 The BeanContext as a Service Provider

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
    );
}
```

The *BeanContextServiceProvider* interface is defined as follows:

```
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
     );
}
```

The *BeanContextServicesListener* is defined as follows:

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

```
}
```

The *BeanContextServiceAvailableEvent* is defined as follows:

```
public class BeanContextServiceAvailableEvent
       extends BeanContextEvent {

     public BeanContextServiceAvailableEvent(
               BeanContextServices        bcs,
               Class                      sc
     );

     BeanContextServices getSourceAsBeanContextServices();

     public Class getServiceClass();

     public boolean isServiceClass(Class serviceClass);

     public Iterator getCurrentServiceSelectors();
}
```

The *BeanContextServiceRevokedEvent* is defined as follows:

```
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();
}
```

The *BeanContextServiceProviderBeanInfo* is defined as follows:

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

Apart from providing a structured hierarchy, the other major role of a *BeanContext* is to provide a standard mechanism for a JavaBean component to obtain context-specific facilities or services from its environment.

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.

A *BeanContextServiceProvider*, is a "factory" for one or more services. It registers itself with a particular *BeanContextServices* via it's *adService*() method, if the service is not already registered with the *BeanContextServices*, the *BeanContextServices* associates the service specified with the *BeanContextServiceProvider,* and fires a *BeanContextServiceAvailableEvent* via the *serviceAvailable*() method to those *BeanContextServicesListeners* currently registered, then returns `true`, otherwise `false` indicating that the service is already registered for that *BeanContextServices*.

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

The *hasService*() method may be used to test the presence of a particular service, and the *getCurrentServices*() method returns an iterator over the currently available services for that *BeanContextServices*.

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 <u>per requestor and service class</u> 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 *BeanContextServices* passes this *getService*() invocation onto the associated *BeanContextServiceProvider* (if any) to be satisfied via an invocation of its *getService*()

method. The *BeanContextServiceProvider* is passed the *BeanContext*, the *Class* of the service provided, the service dependent service parameter (the Service Selector) and a reference to the object requesting the service.

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 Service Selector parameter is a service dependent value used by a service requestor for a particular service in order to parameterize the service to be provided to it by the *BeanContextServiceProvider*. Some examples of its usage are; a parameter to a constructor for the service implementation class; a value for a particular service's property, or as a key into a map of existing implementations.

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.

The *BeanContextServiceProvider* may satisfy the request, returning a reference to an instance of the *Class* of the requested service (such that the reference returned shall result in the expression: <*serviceRefence*> `instanceof` <*serviceClass*> being `true`), return `null`, or throw an unchecked exception.

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.*

A *BeanContextChild* may query a particular *BeanContextServices* for a list of currently available Service Classes (via the *getCurrentServiceClasses*() method)and any associated Service Selectors, if a particular service *Class* implements a finite list of apriori values for a Service Class, via its nesting *BeanContextServices.getCurrentServiceSelectors*() method, which in turn obtains the currently available Service Selectors (if any) via the *BeanContextServiceProvider.getCurrentServiceSelectors*() method.

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`.

A reference obtained by a *BeanContextChild* via *getService*() is valid until the reference is released by the *BeanContextChild* via an invocation of its nesting *BeanContextServices releaseService*() method, except in the case where the *BeanContextServices* fires a *BeanContextServiceRevokedEvent* and that Event's *isCurrentServiceInvalidNow*() method returns `true`, in this case the *BeanContextServices* and/or the *BeanContextServicePro-*

vider that provided the service has determined that current service references are immediately invalidated, or “forcibly revoked” (this typically occurs in the following situation).

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.

BeanContextChild instances receiving a “forcable revocation” of a Service Class shall not invoke *releaseService*() for any references it may hold of that type, since in this case, the *BeanContextServiceProvider* or the *BeanContextServices* that provided the service reference to that *BeanContextChild* has already invalidated all references to that service on their behalf.

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.

Note that in order to function correctly (when delegating service requests) in a multi-threaded environment, implementations of *BeanContextServices* are required to synchronize their implementations of; *addService(), hasService(), getCurrentServiceClasses(), getCurrentServiceSelectors(), getService(), releaseService()* and *revokeService()* with the *BeanContext.globalHierarhyLock.*

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.

2.1.4 The role of a BeanContext in Persistence

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*).

In particular *BeanContext*s shall persist and restore their current children that implement the appropriate persistence interfaces when they themselves are made persistent or subsequently restored.

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*.

BeanContexts shall, when restoring an instance of *BeanContextChild* from its persistence state, be required to perform the equivalent of invoking `add()` on the newly instantiated *BeanContextChild,*, in order to notify the newly restored instance of its nesting *BeanContext*, thus allowing that *BeanContextChild* to fully reestablish its dependencies on its environment.

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

2.1.5 BeanContext with associated presentation hierarchies

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[1] but may be associated with one by implementing the *BeanContextProxy* interface described herein.

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

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

For instances of classes that do not (or cannot in the case of *Component* or subclasses thereof) directly implement the *BeanContext* interface, but are associated with an instance of such an implementation, (via delegation) such instances may expose this association by implementing the *BeanContextProxy* interface. By doing so, this enables arbitrary 3rd parties, such as builder tools, to interrogate and discover The *BeanContext* associated with such objects for the purposes of either nesting objects within that associated *BeanContext*, observing changes in the membership, or obtaining services thereof.

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]

The value returned from *getBeanContext()* is constant for the lifetime of the implementing instance, that is the relationship between a *BeanContextProxy* and it's associated *BeanContext* is static and thus may not change for the lifetime of either participant.

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

Some *BeanContextProxy* implementors may also implement *java.util.Collection*, or some other collection-like API (e.g *java.awt.Container*), in addition to, and possibly distinct from, maintaining a *BeanContext* based *Collection*.

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).

It is important to note that if an object that implements *BeanContextProxy* is added to , or removed from, a *BeanContext*, that in addition to the operation performed on that object, the same operation should be performed on the *BeanContext* returned from *BeanContextProxy.getBeanContext()*. That is an implementor of *BeanContextProxy* shall be treated as though it directly implemented *BeanContext* by any nesting *BeanContext*. (and vica-versa if the operation is applied to the *BeanContext* its shall also be applied to the corresponding *BeanContextProxy*)

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()
}
```

When a *BeanContextChild* with an associated *Component* is added to a *BeanContext* with an associated *Container* there are three models of interaction that can occur in relation to the nesting of the *Component* in the *Container* as a result:

1. 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.
2. 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
3. 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*.

Thus, for greatest interoperability a *BeanContextChild* shall always check if its *Compoent*'s parent is the *BeanContext* Container, and if it is not, then it may add itself if appropriate. Thus a *BeanContextChild* may function correctly under all scenarios.

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].

The nesting *BeanContext*, or its associated *Container*, may subsequently *hide*() or *show*() the *BeanContextChild*'s *Component* arbitrarily, but it is strongly recommended that it treat that *Component* as immutable in all other respects with the exception of registering *Listeners* to obtain event notifications, or where other *Component*/*Container* specific protocols permit or require the *Container* to alter the state of its *Component* containees. An example of such a permitted interaction would be where a property such as *background* or *foreground* color were propagated from *Container* to *Component*.

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).

To avoid infinite recursion, both a *BeanContext* and a *BeanContextChild* that also are associated with a *Container* and *Component* nesting relationship should avoid undoing any changes applied to the *Component* by the other party in the relationship. In general the

BeanContext is responsible for the appearance, visibility and relative layout of the *BeanContextChild*'s *Component,* and the *BeanContextChild* is responsible for the *Component*'s state and content pertaining to the application functionality it is implementing.

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.

In addition the following interface is also defined:

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

A *BeanContext* or a *BeanContextChild* may implement this interface to expose the GUI *Component* that it is associated with to it's nesting *BeanContext*. A *BeanContext* may use this method to establish the relationship between references to instances of *Component* and *BeanContextChild* that are known to it, where a *BeanContextChild* and *Component* are not implemented by the same object instance (that is the *BeanContextChild* delegates its *Component* implementation to a distinct object rather than inheriting from *Component*]. A *BeanContext* may interrogate the *Component* reference it obtains from a nested *BeanContextChild* in order to determine its state, and it may also register *Listeners* for particular events, however it is strongly recommended that the *BeanContext* treat the reference as generally immutable to avoid changing the *Component* state.

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

In the situation where a *BeanContext* has an associated *Container,* but does not wish to expose that *Container* by implementing the *BeanContextContainerProxy* interface, but wishes to handle the nesting of an arbitrary *BeanContextChild*'s associated *Component* (exposed by the *BeanContextChild* either implementing the *BeanContextChildComponentProxy* interface or as direct subclass of *Component*) the *BeanContext* is permitted to add/remove that *Component* to/from its associated *Container*.In such cases the *BeanContextChild* and it's associated *Component* implementation shall not interfere with this action.

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

2.2 interface java.beans.beancontext.BeanContextChild[1]

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

1. I don't like this name much but I am struggling for a better alternative! (we are stuck with it)

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);

}
```

The expected usage is that some 3rd party shall invoke one of the appropriate methods defined on *BeanContext* (by virtue of its inheritance from *Collection*) in order to add a *BeanContextChild* to the membership of the target *BeanContext*. As a consequence the *BeanContext* shall attempt to set the *BeanContextChild*'s "beanContext" property by invoking its setter method, *setBeanContext*(). Only a *BeanContext* may call a *BeanContextChild*'s *setBeanContext*() method, since this is the mechanism that a *BeanContext* uses to notify a child that it is now has a new *BeanContext* value. Since this property is not directly settable or customizable by a user in the context of an application construction tool the *BeanInfo* for a *BeanContextChild* should set the hidden state for this property in order for builder tools to avoid presenting the property to the user for customization.

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.

During the un-nesting of a *BeanContextChild* from its *BeanContext*, it is possible for the child, or a 3rd party listening to the child's "beanContext" property for *PropertyVetoEvents*, to throw a *PropertyVetoException* to notify the caller that it is not in a state to be un-nested. In order to bound this interaction a *BeanContextChild*, or 3rd party, may veto the initial un-nesting notification, but may not veto any subsequent notifications, and must, upon receipt of such notifications, amend its state accordingly to prepare itself to be subsequently un-nested.

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.

BeanContextChild instances, or nested *BeanContext*s in the process of terminating themselves, shall invoke the *remove()* method on their nesting *BeanContext* in order to withdraw themselves from the hierarchy prior to termination.

2.2.1 Important Persistence considerations

Instances of *BeanContextChild* nested within an *BeanContext,* will typically define fields or instance variables that will contain references to their nesting *BeanContext* instance, and possibly services obtained from that *BeanContextServices* instance via its *getService()* method.

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.

This requirement is crucial since operations such as cutting and pasting object instances through a clipboard via object serialization will not function correctly if the act of serializing the target object also serializes much of the entire source environment it is nested within.

3.0 Overloading java.beans.instantiate() static method

Since *java.beans.instantiate()* is the current mechanism for (re)instantiating JavaBeans we need to extend or overload the syntax and semantics of this method in order to accommodate the introduction of the *BeanContext* abstraction. The extension proposed is:

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

This method behaves has it is currently defined in the JavaBeans specification, but in addition to these existing semantics, when a non-`null` *BeanContext* is specified then the method invokes the *add()* method on the *beanContext* actual parameter with the value of the *targetChild* actual parameter = a reference to the newly instantiated JavaBean component.[1]

4.0 Providing better support for Beans that are also Applets

The current implementation of *java.beans.instantiate()* contains minimal support for instantiating JavaBeans that are also Applets. In particular, this method will currently construct an *AppletContext* and *AppletStub* for the newly instantiated JavaBean, set the stub on the newly instantiated *Applet,* and *init()* the *Applet* if it has not already been invoked.

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.

Since key to supporting fully functional Applets is to provide them with fully functional *AppletContext* and *AppletStub* instances, the design goal is to provide a mechanism to provide this state to *instantiate()* so that it may carry out the appropriate initialization and binding[1], therefore the proposed interface is:

```
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);
}
```

If the newly instantiated JavaBean component is an instance of *java.applet.Applet* then the new constructed *Applet*, (Bean) will be passed to the *AppletInitializer* via a call to *initialize().*

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

1. Associate the newly instantiated *Applet* with the appropriate *AppletContext*.

---

1. Note: 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 themselves from the hierarchy and thus subsequently be garbage collected.

1. *AppletContext* objects expose a list of *Applet* objects they "contain", unfortunately the current *Applet* or *AppletStub* API's as defined, provide no mechanism for the *AppletContext* to discover its *Applets* from its *AppletStubs,* or for an *AppletStub* to inform its *AppletContext* of its *Applet.* Therefore we will have to assume that this binding/discovery can occur in order for this mechanism to be worthwhile in *java.beans.instantiate()*.

2. Instantiate an *AppletStub*() and associate that *AppletStub* with the *Applet* via an invocation of *setStub*().
3. 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.

Compliant implementations of *AppletInitializer.activate()* shall mark the *Applet* as active, and may optionally also invoke the *Applet*'s *start()* method.

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

5.0 Standard/Suggested Conventions for BeanContext Services

5.0.1 BeanContexts that support InfoBus.

The InfoBus technology is a standard extension package that is intended to facilitate the rendezvous and exchange of dynamic self describing data, based upon a publish and subscribe abstraction, between JavaBean Components within a single Java Virtual Machine.

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.*

Thus *BeanContextChild* implementations may locate a common *InfoBus* implementation for their current *BeanContext* by using this mechanism to rendezvous with that *InfoBus* instance.

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*.

5.0.2 BeanContexts that support printing

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

As the Java Network Printing Interface evolves additional specifications will be provided to expose it's facilities via the services mechanism.

5.0.3 BeanContext Design/Runtime mode support.

JavaBeans support the concepts of "design"-mode, when JavaBeans are being manipulated and composed by a developer in an Application Builder or IDE, and "Run"-mode, when the resulting JavaBeans are instantiated at runtime as part of an *Applet*, Application or some other executable abstraction.

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.

Therefore we require the ability to scope this “mode” at a granularity below that of JVM global.

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

Thus *BeanContext*’s that wish to expose and propagate this “mode” to its descendants may delegate a reference of type *java.beans.DesignMode*:

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

Additionally, *BeanContext*s delegating such a reference shall be required to fire the appropriate *java.beans.propertyChangeEvent,* with propertyName = “designTime”, with the appropriate values for *oldValue* and *newValue,* when the “mode” changes value.

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

5.0.4 BeanContext Visibility support.

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).

The first version of the JavaBeans Specification introduced the *java.beans.Visibility* interface in order to provide a mechanism for JavaBeans to have their “visible” state, or ability to render a GUI, controlled from their environment.

*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.

5.0.5 Determining Locale from a BeanContext

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

Therefore, *BeanContext*s, shall be required to fire the appropriate *java.beans.PropertyChangeEvent*, with propertyName = “locale”, *oldValue* = the reference to the previous value of the *Locale* delegate, and *newValue* = the reference to the new value of the *Locale* delegate, in order to notify its Listeners of any change in *Locale*.

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

6.0 Support classes

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.