Controls Programming

Table of contents

1. Overview

The Control architecture is a lightweight component framework based upon JavaBeans, which exposes a simple and consistent client model for accessing a variety of resource types. Controls take the base functionality of JavaBeans and add in the following unique new capabilities:

- Enhanced authoring model: uses a public interface contract and an associated implementation class to enable generation of a supporting JavaBean class for handling the details of property management and initialization.
- Extensibility model: enables the construction of views and custom operations (with implied semantics) on the Control using metadata-annotated interfaces.
- metadata attributes and external configuration data: provides an enhanced configuration model for resource access.

This document focuses on the Controls programming and configuration model from two distinct perspectives:

- The authoring and extensibility model for defining a new type of Control
- The client access model for declaring and using Controls

An overview of the Control architecture and toolable access models can be found in the companion document entitled Control Overview: Providing Simplified and Unified Access to J2EE Resources

2. An Example

In the course of describing the programming model for Controls, this document builds upon an example Control that simplifies the enqueueing of JMS messages with a specific format and set of properties. Once completed, client code to accomplish this should be as straightforward as:

Enqueueing using OrderQueueBean (*Client Code*)

```
OrderQueueBean orderBean = (OrderQueueBean)

java.beans.Beans.instantiate("org.apache.beehive.controls.examples.OrderQueueBean");
Order order = new Order(myID, new String [ ] {"item1", "item2"});
OrderBean.submitOrder(order, "01-28-2004");
```

This document starts with a brief overview of the Control Authoring and Client Programming Models to establish some basic context, eventually building to enable the example above.

3. The Control Authoring Model

This section describes the basic authoring model for Controls. This includes a description of the following elements:

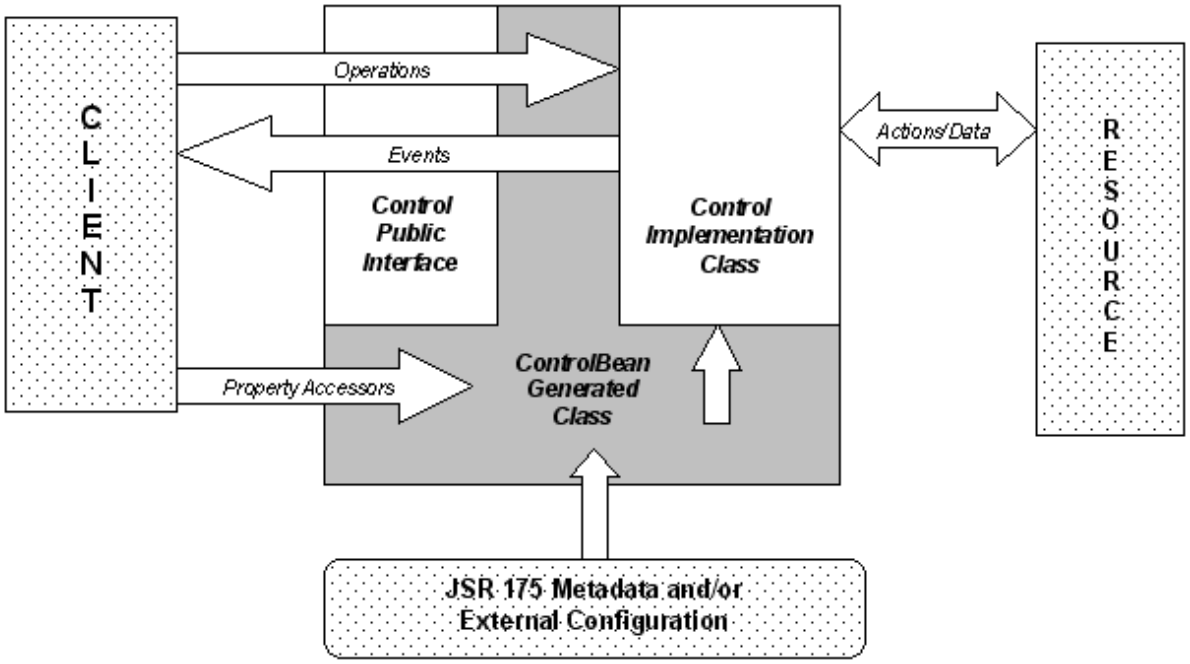
- ***Control Public Interface***: source file that defines the set of operations, events, extensibility model, and properties associated with the Control type.
- ***Control Implementation Class***: source file that provides the implementation of the operations and extensibility model described by the Control Public Interface.
- ***ControlBean Generated Class***: code-generated JavaBean class that is derived from the Control Public Interface and the Control Implementation Class by a Control compiler.

This authoring model is a departure from the traditional JavaBeans programming model, which is largely based upon a set of conventions that a bean author is expected to follow when constructing a new JavaBean type. In the Controls model, the author defines operations, events, and properties in an interface (Control Public Interface) and builds an underlying implementation (Control Implementation Class). A Control compiler takes these two elements and generates a specialized type of JavaBean (ControlBean Generated Class), which represents the full client programmer's view of the Control.

There are two primary advantages of this model:

- **Simplicity.** A key goal of any ease-of-use programming model is to free the developer from worrying about plumbing. Managing property values, event listener lists, and other basic JavaBean functions are fairly rote from implementation to implementation. The Controls architecture employs a unique variant of the Inversion of Control (IoC) design pattern based on metadata annotations. This enables a Control Implementation Class to declaratively specify the events or services it requires to provide its semantics. The ControlBean Generated Class acts as a lightweight container to provide contextual hookup and implementation details.
- **Consistency.** Instead of trying to provide consistency through convention, the Control compiler provides both verification and code generation services to ensure that the resulting implementation provides consistent APIs and behaviors for clients, tools, and application deployers or administrators.

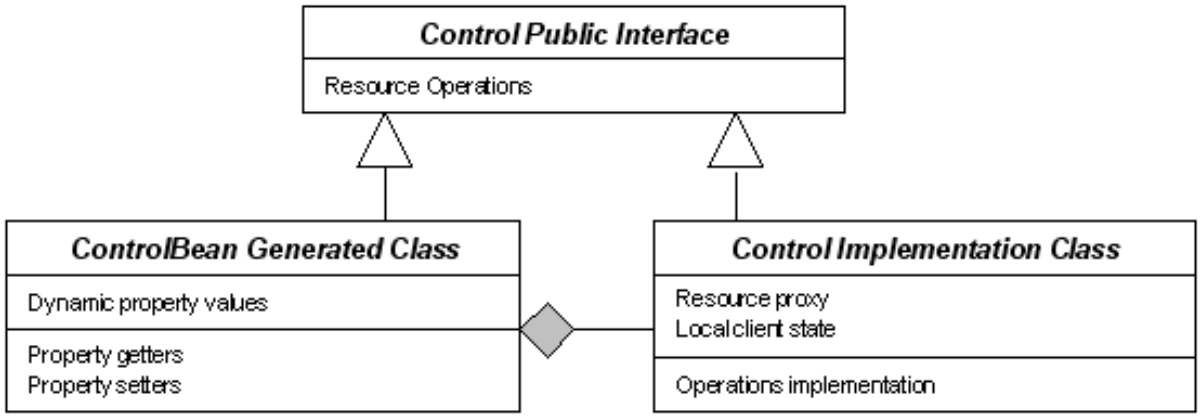
Diagram: Control Architecture Elements and Flow



The client will interact with the Control by invoking operations defined on the Control Public Interface or dynamic property accessor methods on a ControlBean instance. The client can also express interest in any events the Control might generate by registering a listener to receive them.

The following diagram represents the relationship between the Control Public Interface, the Control Implementation Class, and the ControlBean Generated Class:

Diagram: Relationships between Control Interface and Classes



The Control Public Interface defines the operations on the Control and will be implemented by both the Control Implementation Class and the ControlBean Generated Class. The ControlBean Generated Class will also define property accessor methods and internally will maintain the state of property values. It will also maintain a reference to one (and only one) Control Implementation instance. The Control Implementation instance, wrapped by a bean instance, provides the actual implementation of resource semantics for the Control.

The subsequent sections will outline the various characteristics of Controls:

- Declaration / Instantiation
- Operations
- Events
- Contextual Services
- Properties
- Extensibility
- Composition
- Context Events

Where applicable, the aspects of each of these characteristics will be explored in two dimensions: from the perspective of a Control author who is defining a new type of Control, and from the perspective of a Control client that is using the services of an available Control type.

To make the descriptions more concrete, the characteristics will be presented within the context of a sample Control: the JmsMessageControl. This Control will provide a simplified client access model for enqueuing messages to a JMS queue or topic, freeing the client from having to learn the nuances of JMS client programming.

4. The Control Client Models

There are actually two distinct programming models that may be available to clients of Controls:

- **Declarative Model.** Uses a metadata-based variant of the Inversion of Control (IoC) design pattern to allow a component author to declare Control instances, contextual services, and event handlers using annotated fields and methods. The declarative model simplifies client programming, because many of the details of initialization and event routing are left to an external container supporting the model. A declarative programming style is also more toolable, since it is much easier for tools to manage and manipulate metadata rather than code.
- **Programmatic Model.** Uses the traditional JavaBean-style APIs for acting as a client of a bean, including factory-based constructor and event listeners. The programmatic model

may be more comfortable to the traditional Java programmer, who wants to see and be in control of all the details. It also enables client use cases where there is no supporting container for the declarative model.

The programmatic client model is generally available in all contexts where Controls might be used. It offers full generality, but leaves many of the details up to the client programmer, such as initialization, composition, and event handling wire-up.

The declarative model hides many of these details. Based upon its use of metadata it is also more tool friendly, allowing tools to present a view of the client code without code analysis.

The declarative model requires support of an outer container or construction-time code that fulfills the contract implied by annotations on a client class.

The ControlBean itself provides this support, so the Control Authoring Model is oriented towards using the declarative model, although programmatic equivalents are generally available.

5. Defining a New Control Type

Controls are designed to make it very easy for users (and tools) to define new types of Controls. Control authors might be:

- System vendors exposing specific types of resources
- Application developers defining new types of logical resources (possibly based upon physical ones)
- Third-party software vendors, using Controls as a mechanism to interface to components or subsystems they provide.

In all instances, the goal of the Controls authoring model is to provide a basic set of conventions and supporting tools to make it easy to author a new Control type.

To get started, a Control author would define the two basic artifacts:

- the Control Public Interface
- the Control Implementation Class

For the JmsMessageControl, the declaration of the public interface might look like:

Interface Declaration (Control Public Interface)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlInterface;

@ControlInterface
public interface JmsMessageControl
```

```
{
    ...
}
```

The only basic rule for a Control Public Interface is that it must be annotated by the org.apache.beehive.controls.api.bean.ControlInterface marker interface.

The second source artifact a Control author would create to define a new type of Control is the Control Implementation Class. This declaration of the implementation class for our JmsMessageControl would look like:

Class Declaration (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlImplementation;

@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl
{
    ...
}
```

The basic rules for a Control Implementation Class are:

- It must be annotated with `org.apache.beehive.controls.api.bean.ControlImplementation.`
- It must implement its associated Control Public Interface.
- It must either (1) implement the `java.io.Serializable` interface or (2) set the `isTransient` attribute of `@ControlImplementation` to true.

From these two source files, the Control compiler will create a third artifact, the ControlBean Generated Class. This class need not necessarily ever appear within an application in source code form; but for the purposes of explaining the overall architecture and client model, we will present source examples of the derived ControlBean Generated Class.

A Controls standard would focus only on the conventions for the external attributes of ControlBean Generated Classes, not upon the internal implementation.

The ControlBean Generated Class for the JmsMessageControl would look like:

Class Declaration (ControlBean Generated Class)

```
package org.apache.beehive.controls.examples;

public class JmsMessageControIBean implements JmsMessageControl
{
   private JmsMessageControlImpl _impl;

    ...
}
```

As shown above, the ControlBean Generated Class will also implement the Control Public Interface. The sample also shows that the bean will hold a private reference to an implementation instance used to support the bean.

6. Instantiating a Control

This section covers the client mechanisms for creating a new instance of a Control. This can be done either programmatically or declarative, if running inside a container that support declarative initialization.

6.1. Declarative Instantiation

The client model for Controls supports a declarative model for instantiating a Control instance, when running in containers that support this model. In this model, the client class can annotate fields on the class using a special marker annotation (org.apache.beehive.controls.api.bean.Control) that indicates that the fields should be initialized to a ControlBean instance of the requested type.

Here is an example of declarative instantiation:

Declarative Instantiation (Client Code)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.Control;
import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.examples.JmsMessageControl;

@ControlImplementation(isTransient=true)
public class PublisherControlImpl implements PublisherControl
{
    @Control
    public JmsMessageControlBean myJmsBean;

    ...

    public void someOperation()
    {
        myJmsBean.sendTextMessage("A Text Message");
    }
}
```

This example shows a second Control Implementation Class (PublisherControlImpl) that internally uses the services of JmsMessageControl to enqueue a JMS message. The child Control field is not explicitly initialized within the PublisherControl implementation class; by the time someOperation() is called, it is guaranteed that the myJmsControl reference has been initialized by the wrapping PublisherControl that contains the implementation.

It is also possible to parameterize a Control at construction time, again using metadata attributes. These attributes can be placed on the field declaration (in addition to the @Control annotation) and will be used to do construction-time initialization.

The second example below shows initialization of the myJmsControl field again. In this case, an initial value of the @Destination "name" attribute is also provided using metadata annotations:

Declarative Instantiation with Properties (Client Code)

```
@ControlImplementation
public class PublisherControlImpl implements PublisherControl
{
    @Control
    @Destination(name="InvoiceQueue")
    public JmsMessageControlBean myJmsBean;
```

This example performs **exactly** the same initialization as the earlier declarative example, but does so using annotation attribute syntax instead of passing parameters to a factory-based constructor.

The Controls architecture includes a mechanism for defining the expected set of annotations that might appear on a Control field. This mechanism is described in greater detail in the section on Properties.

6.2. Programmatic Instantiation

The client model for Controls supports instantiation of a new Control instance using the same factory-based model supported by JavaBeans. For example, the following code could be used to create a new instance of the JmsMessageControlBean generated class:

Programmatic Instantiation (Client Code)

```
JmsMessageControlBean myJmsBean =
   (JmsMessageControlBean) java.beans.Beans.instantiate(
       cl, "org.apache.beehive.controls.examples.JmsMessageControlBean"
   );
```

The Control runtime also provides an extended factory model that allows metadata attributes to be passed into the factory constructor:

Programmatic Instantiation with Properties (Client Code)

```
import org.apache.beehive.controls.api.bean.Controls;
import org.apache.beehive.controls.api.properties.PropertyKey;
import org.apache.beehive.controls.api.properties.PropertyMap;
import org.apache.beehive.controls.api.properties.BeanPropertyMap;

PropertyMap jmsAttr = new BeanPropertyMap(JmsControl.Destination.class);
jmsAttr.setProperty(new PropertyKey(JmsControl.Destination.class, "name"),
```

```
"InvoiceQueue");
JmsMessageControlBean myJmsBean = (JmsMessageControlBean)
Controls.instantiate(
    cl, "org.apache.beehive.controls.examples.JmsMessageControlBean",
jmsAttr
);
```

In this example, the JmsMessageControlBean is being constructed with the Destination "name" property set to "InvoiceQueue". The AttributeMap class is a simple helper class that can hold a set of name-value pairs of a Control's properties, which are initialized by the factory-based constructor. More details on Controls properties are provided in a later section.

7. Operations

Operations are actions that can be performed by a Control at the client's request. This section describes the authoring model for declaring and implementing a Control operation, as well as the client model for invoking operations on a ControlBean instance.

7.1. Declaring and Implementing Operations for a Control

All methods declared or inherited (via extension) by the Control Public Interface are considered to be Control operations. The following example shows the definition of two operations on the JmsMessageControl that will enqueue messages when invoked:

Declaring Operations (Control Public Interface)

```
package org.apache.beehive.controls.examples;

import java.io.Serializable;
import org.apache.beehive.controls.api.bean.ControlInterface

@ControlInterface
public interface JmsMessageControl
{
    public void sendTextMessage(String text);
    public void sendObjectMessage(Serializable object);

    ...
}
```

The Control Implementation Class implements the public interface for the Control, defining the operation methods, and the body of these methods.

Implementing Operations (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

import java.io.Serializable;
import org.apache.beehive.controls.api.bean.ControlImplementation;
```

```
@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl
{
   public void sendTextMessage(String text)
    {
        // Code to send a TextMessage to the destination
        ...
    }

   public void sendObjectMessage(Serializable object)
    {
        // Code to send an ObjectMessage  to the destination
        ...
    }
}
```

Finally, the ControlBean Generated Class will also implement all operations (since it also implements the Control Public Interface). It will always delegate to the implementation class for the actual implementation of the operation; it might also perform additional container-specific pre/post invocation processing.

Here is a skeleton of what the generated ControlBean code might look like for an operation:

Implemented Operations (ControlBean Generated Class)

```
package org.apache.beehive.controls.examples;

public class JmsMessageControlBean implements JmsMessageControl
{
   private JmsMessageControlImpl _impl;

     public void sendTextMessage(String text)
    {
        ...
        _impl.sendTextMessage(text);
        ...
    }

    public void sendObjectMessage(Serializable object)
    {
        ...
        _impl.sendObjectMessage(object);
        ...
    }
```

7.2. Invoking Operations on a Control

The client model for invoking an operation on a Control is very straightforward: simply call the method on a held ControlBean instance as demonstrated by the following example:

Invoking an Operation (Client Code)

```
package org.apache.beehive.controls.examples;
```

```
import org.apache.beehive.controls.api.bean.Control;
import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.examples.JmsMessageControl;

@ControlImplementation(isTransient=true)
public class PublisherControlImpl implements PublisherControl
{
    @Control
    public JmsMessageControlBean myJmsBean;

    ...

    public void someOperation()
    {
        myJmsBean.sendTextMessage("A Text Message");
    }
}
```

The invocation model for operations is the same, whether the Control instance was created using declarative or programmatic mechanisms.

8. Events

Events are notifications sent by the Control back to its client whenever some condition has been met or internal event has taken place. A client can express interest in a Control's events by registering (either explicitly or implicitly) to receive them, and can write event handler code to be called when the event has taken place.

This section describes the declaration model for events, how an authored Control delivers them to a registered client, and the client code necessary to register and receive events.

8.1. Declaring Events

Events are declared on an inner interface of the Control Public Interface, which is annotated with the org.apache.beehive.controls.api.events.EventSet annotation. The following example shows the declaration of an event interface for the JmsMessageControl, with a single event (onMessage):

Declaring Events (Control Public Interface)

```
package org.apache.beehive.controls.examples;

import java.io.Serializable;
import javax.jms.Message;
import org.apache.beehive.controls.api.events.EventSet;
import org.apache.beehive.controls.api.bean.ControlInterface;

@ControlInterface
```

```
public interface JmsMessageControl
{
    public void sendTextMessage(String text);
    public void sendObjectMessage(Serializable object);

    @EventSet
    public interface Callback
    {
        void onMessage(Message m);
    }

    ...
}
```

If a Control Public Interface has defined an EventSet interface, then the associated ControlBean Generated Class will have two public methods supporting client listener management:

Event Listener Registration Methods (ControlBean Generated Class)

```
package org.apache.beehive.controls.examples;

import java.util.TooManyListenersException;

public class JmsMessageControIBean implements JmsMessageControl
{
   ...

    /** Registers a new client listener for this bean instance */
   public void addCallbackListener(Callback listener)
       throws TooManyListenersException
   {
      ...
   }

   /** Deregisters a client listener for this bean instance */
    public void removeCallbackListener(Callback listener)
    {
         ...
    }
}
```

The name of the listener registration methods are based upon the name of the associated EventSet interface. In the previous example, the EventSet interface was named Callback, so the associated listener registration method was addCallbackListener(), and the deregistration method was removeCallbackListener().

A Control Public Interface can have more than one inner interface that is annotated as an EventSet interface. Each declared EventSet will have its own independently managed list of registered listeners.

8.2. Firing Events

This section describes the mechanism available to a Control author to deliver events to any registered client listener. **An initialized event proxy is created when the Control Implementation Class declares a field of an EventSet interface type, and annotates it with the org.apache.beehive.controls.events.Client annotation type.** The containing ControlBean will initialize this reference to a valid proxy implementing the EventSet interface, and the Control Implementation Class can use this proxy to fire events back to any registered client.

This is demonstrated in the following sample code from the JmsControlBean implementation class, which will fire an onMessage event back to any registered client any time a message is enqueued:

Firing Events (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

import java.io.Serializable;
import javax.jms.TextMessage;
import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.api.events.Client;

@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl
{
    @Client Callback client;

    public void sendTextMessage(String text)
    {
        // Code to construct and send a TextMessage to the destination
       TextMessage m = ...;
        ...
        client.onMessage(m);
    }
    ...
}
```

8.3. Listening for Events

The client of a Control can express an interest in receiving events from a Control and write client event handlers to service them once delivered. Two basic event handling mechanisms are supported: Java event listeners or declarative event handlers (where supported by the client container).

8.3.1. Declarative Implementation of Event Handling

If the client code is implemented in a container that supports the declarative programming model for Controls (such as the Control Implementation Class itself), it can use a simplified convention for authoring event handlers for a declared Control instance.

If a Control is declared using the @Control marker interface, then **the user can declare event handlers for the Control by using the EventHandler annotation type**. These annotated methods will be considered an event handler for the Control event, and the container will automatically register for events and deliver them to this handler.

The previous example could be rewritten using the declarative event handling style as:

Declarative Handling of Events (Client Code)

```
package org.apache.beehive.controls.examples;

import javax.jms.Message;
import org.apache.beehive.controls.api.bean.Control;
import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.api.events.EventHandler;
import org.apache.beehive.controls.examples.JmsMessageControl;

@ControlImplementation(isTransient=true)
public class PublisherControlImpl implements PublisherControl
{
    @Control
    public JmsMessageControlBean myJmsBean;

    @EventHandler (
        field="myJmsBean",
        eventSet= JmsMessageControl.Callback.class,
        eventName="onMessage")
     public void myJmsBeanMessageHandler(Message m)
     {
        // Code implementing onMessage event handler
     }
    ...
}
```

8.3.2. Programmatic Implementation of Event Handling

The programmatic style follows the traditional Java event listener pattern. The client expresses its interest in receiving the event and also authors a (often anonymous inner) class that implements the event interface to receive events when delivered.

This is shown by the following sample code:

Programmatic Handling of Events (Client Code)

```
myJmsBean.addCallbackListener(
    new JmsMessageControl.Callback()
    {
```

```
        public void onMessage(Message m)
        {
            // Code implementing on Message event handler
        }
    }
);
```

There is no requirement that an anonymous inner class be used. One alternative would be to delegate to an instance of another class (as long as that class implements the Callback interface). In the preceding example, if event listening was implemented for the purposes of logging sent messages, and MessageLogger class could be declared (implementing the Callback interface), multiple beans could delegate to a single instance of this logging listener.

9. Contextual Services

The Control authoring model makes use of contextual services to provide access to services from the current runtime environment of the ControlBean. The model for contextual services is based upon the existing standards for services in JavaBeans: The JavaBeans Runtime Containment and Services Protocol. This protocol provides a base mechanism for a JavaBean to locate and use services from the runtime environment, as well as an extensible service provider model to enable new (or environment-specific) types of services to be authored and made available to JavaBeans/Controls.

A key aspect of this service model is that it can be contextual; for example, it might be possible to write a basic security service interface that provides logical role-checking functionality. The actual implementation of this interface might vary for different runtime contexts: for example, the role check might be done differently for a Control running within the context of an EJB container (by delegating to the containing EJBContext) vs. a Control running within the Web tier (by delegating to ServletHttpRequest services).

Having an extensibility and service provider location model is important to enable the following scenarios:

- The Control's implementation is designed to run in a wide variety of environments. It uses the contextual service mechanism to declare its prerequisites and receive a provider implementation that is appropriate to the current runtime context.
- The Control's implementation is designed to run in a very specific context (for example, only in the http servlet tier) and wants access to services that are very specific to that context (for example, session state or request query parameters). It should not be possible to instantiate this Control in other contexts (for example, from within an EJB).

One key contextual service for Controls that is guaranteed to be available in all contexts is the org.apache.beehive.controls.api.context.ControlBeanContext service interface. This service provides a common set of generic services that are available to Control authors,

such as the ability to query property values on the current instance, or to receive a set of basic lifecycle or resource management events. The ControlBeanContext interface extends the java.beans.beancontext.BeanContextServices interface, so it also provides access to services provided by the JavaBeans bean context APIs. Later sections describe an overview of the internal architecture for contextual services, APIs to support property resolution, and lifecycle events.

9.1. Declarative Access to Contextual Services

Suppose the following Destination property set was added to the control:

Declarative Access to Context Services (Control Public Interface)

```
package org.apache.beehive.controls.examples;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.apache.beehive.controls.api.bean.ControlInterface;
import org.apache.beehive.controls.api.properties.PropertySet;

@ControlInterface
public interface JmsMessageControl
{
    ...

    public enum DestinationType { QUEUE, TOPIC }

    @PropertySet
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.FIELD, ElementType.TYPE})
    public @interface Destination
    {
        public DestinationType type() default DestinationType.QUEUE;
        public String name();
    }
}
```

To signal the desire to access a contextual service, a Control author only needs to declare a field of the desired context interface and annotate it with the org.apache.beehive.controls.api.context.Context marker annotation. The following example shows how the JmsMessageControlImpl class would use the declarative model to access its ControlBeanContext:

Declarative Access to Context Services (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlImplementation;
```

```
import org.apache.beehive.controls.api.context.Context;
import org.apache.beehive.controls.api.context.ControlBeanContext;

@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl
{
    @Context ControlBeanContext context;

    public void sendTextMessage(String text)
     {
        JmsMessageControl.Destination destProp =
            context.getControlPropertySet(JmsMessageControl.Destination.class);
         ...
     }
}
```

In this example, the JmsMessageControl implementation class expresses its desire to access ControlBeanContext services via the annotated declaration of the context field; when code in sendTextMessage operation is invoked, this contextual service has already been initialized by the containing ControlBean instance.

The ControlBeanContext for an authored Control is always accessed using the declarative mechanism. Other contextual services may be accessed declaratively, or using the programmatic mechanisms described in the following section.

9.2. Programmatic Access to Contextual Services

The ControlBeanContext service also provides the base mechanism to discover and use other services programmatically. The following code fragment shows an example of how to use this API to obtain access to a service provider that provides the javax.servlet.ServletContext interface.

Programmatic Access to Context Services (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

import javax.servlet.ServletContext;
import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.api.context.Context;
import org.apache.beehive.controls.api.context.ControlBeanContext;

@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl
{
    @Context ControlBeanContext context;

    public void sendTextMessage(String text)
    {
        ServletContext servletContext =
            context.getService(ServletContext.class, null);
        if (servletContext == null)
```

```
        {
            //  no ServletContext provider is available
        }
        ...
    }
}
```

The code in the sample uses the ControlBeanContext.getService API to request that it provide a ServletContext service. The parameters to this method are the Class of the requested service, and an (optional) service-specific selector that can be used to parameterize the service.

The ServletContext service is contextual because it is available only to controls running in the web tier. If the above sample control was running anywhere else, the call to ControlBeanContext.getService() would return null.

9.3. Tradeoffs between Declarative and Programmatic Access

Declarative access to context services is always available to a Control Implementation Class, and generally results in less code associated with accessing services. Why then, would using programmatic access ever be useful? There is a key difference between the two:

- When using the declarative model for accessing a contextual service, the Control is effectively saying that the service is required for it to function; if not available in a particular runtime environment, then construction of an instance of the Control will fail. Essentially, the annotated context acts as a notification to the runtime factory that this prerequisite must be satisfied.
- Use of the programmatic model allows a Control Implementation Class to implement conditional behavior based upon whether a contextual service is or is not available. The Control Implementation Class can use the programmatic accessor, and then make a decision how to proceed based upon whether the requested service is available.

10. Properties

This section describes Control properties. Properties provide the basic mechanism for parameterizing the behavior of a Control instance.

The Controls architecture takes the basic JavaBeans notion of properties and extends it to support two new capabilities:

- A declarative annotation model where properties can be preconfigured on a ControlBean using metadata annotations
- An administrative model where the value of ControlBean properties can be externally defined or overridden.

The external configuration and administrative model for Controls will be described in a separate document.

10.1. Declaring Properties for a Control Type

For Controls, the set of properties is explicitly declared on the Control Public Interface. This makes the available parameterization of a Control type readily visible to both code and tools.

Properties are grouped together into related groups called PropertySets. All Properties within a PropertySet will have a common set of attributes (such as where they can be declared, the access model for JavaBean accessors, etc) and will have property names based upon a common naming convention.

A PropertySet is declared as a metadata attribute interface within the Control Public Interface, which is also decorated with the org.apache.beehive.controls.api.properties.PropertySet meta-attribute. Each of the members within a PropertySet will refer to a distinct property within the set, and the return value of the member defines the property type.

Here is a sample declaration of the Destination PropertySet for the JmsMessageControl, which can be used to configure the target JMS destination for the Control:

Declaring Properties (Control Public Interface)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlInterface
import org.apache.beehive.controls.api.properties.PropertySet;

import java.lang.annotation.ElementType;
import java.lang.annotations.Retention;
import java.lang.annotations.RetentionPolicy;
import java.lang.annotations.Target;

@ControlInterface
public interface JmsMessageControl
{
     ...

    public enum DestinationType { QUEUE, TOPIC }

    @PropertySet(prefix="Destination")
    @Target({ElementType.FIELD, ElementType.TYPE})
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Destination
    {
        public DestinationType type() default DestinationType.QUEUE;
        public String name();
    }
```

```
    ...
}
```

This declaration defines the PropertySet named 'Destination' that includes two properties: type and name. The type property is based upon the DestinationType enumerated type, which is also defined in the public interface. The name attribute is a simple String property.

Meta-attributes on a PropertySet or property declaration can be used to provide additional details about the properties and how they may be used. In the above example, the standard java.lang.annotations.Target annotation is used to define the places where the @Destination property set can appear (in this case in either an extension class or field declaration).

The full set of meta-attributes that can decorate PropertySet or Property declarations are TBD. They can be used to define constraint models for property values, or relationships between properties (such as exclusive or, where one is set or the other, but never both). These meta-attributes can be read and used by development or administrative tools to aid in the selection of property values. They can also be used by the runtime for runtime validation of property values when set dynamically. The current set of property constraint mechanisms is implemented by the `@AnnotationConstraints` annotation. See Defining Property Constraints below for details.

10.2. Accessing Properties from Client Code

The properties defined in the Control Public Interface will be exposed to the client programmer using traditional JavaBean setter/getter methods on the ControlBean Generated Class. These methods will follow a simple naming pattern based upon the PropertySet interface name, and optional PropertySet prefix, and property member name.

The basic pattern for these accessors is:

Property Accessor Generation (Conventions)

```
public void set<PropertySetPrefix><MemberName>(<MemberType>);
public <MemberType> get<PropertySetPrefix><MemberName>();
```

The PropertySetPrefix refers to the optional prefix attribute of the PropertySet annotation. If unspecified, it will default to an empty string (no prefix). The MemberName refers to the PropertySet method name that declares the property, with the first character converted to uppercase, and the MemberType refers to the return value type of this method declaration.

So for the Destination PropertySet interface shown in the example above, the resulting ControlBean Generated Class would expose the following accessors:

Property Accessors (ControlBean Generated Class)

```
package org.apache.beehive.controls.examples;
```

```
import java.util.TooManyListenersException;

public class JmsMessageControIBean implements JmsMessageControl
{
    ...
    public void setDestinationType(DestinationType type)  { ... }
    public DestinationType getDestinationType() { ...}
    public void setDestinationName(String name) { ...}
    public String getDestinationName();
}
```

Client code to set the Destination properties on a JmsMessageControlBean instance would look like:

Using Property Accessors (Client Code)

```
    @Control JmsMessageControlBean jmsBean;

    ...

    jmsBean.setDestinationType(DestinationType.QUEUE);
    jmsBean.setDestinationName("myTargetQueue");
```

10.3. Accessing Properties from Control Implementation code

The Control Implementation class contains code that executes from within the context of the Control JavaBean that is generated to host the control. The generated bean will automatically manage the resolution of properties values from annotations, external configuration, or dynamic values set by the client.

Access to these properties is provided by the ControlBeanContext instance associated with the Control Implementation Class. This interface provides a set of property accessors that allow the implementation to query for property values:

ControlBeanContext APIs for Property Access

```
package org.apache.beehive.controls.api.context;

public interface ControlBeanContext
    extends java.beans.beancontext.BeanContextServices
{
    ...
    public <T extends Annotation> T
      getControlPropertySet(Class<T> propertySet);
    public <T extends Annotation> T
      getMethodPropertySet(Method m, Class<T> propertySet);
    public <T extends Annotation> T
      getParameterPropertySet(Method m, index I, Class<T> propertySet);
     ...
}
```

The propertySet argument passed to these methods must be a valid PropertySet interface

associated with the ControlInterface. The ControlBeanContext will return the current value for properties in the PropertySet, or will return null if no PropertySet value has been associated with this control instance.

Here is a simple example of using ControlBeanContext.getControlPropertySet() to query a property set:

Acccessing Control Properties (Client Implementation Class)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.api.context.Context;
import org.apache.beehive.controls.api.context.ControlBeanContext;

@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl
{
    @Context ControlBeanContext context;

    public void sendTextMessage(String text)
    {
        ...
        Destination destProperty = (Destination)
context.getControlPropertySet( Destination.class );
        if( destProperty == null ) {
            System.out.println( "Dest Property NOT Set" );
        } else {
            System.out.println( "Dest Property Set" );
        }
   }
}
```

This code above queries for the value of the JmsMessageControl.Destination PropertySet on the current JmsMessageControl instance.

These query methods will return the value of resolved properties for the Control instance, method, or method argument, respectively. Control implementations should never use Java reflection metadata accessors directly on Control classes or methods; these accessors won't reflect any property values that have been set dynamically by ControlBean client accessor methods or externally using administrative configuration mechanisms. The ControlBeanContext provides a consistent resolution of source annotation, client-provided, and external values.

A simple example of using the ControlBeanContext property accessor methods for accessing Method and Parameter properties is provided in the section on Extensibility.

10.4. External Configuration of Control Properties

Controls also support an administrative model that allows Control property values to be bound using external configuration syntax. The enables Control behavior to be parameterized externally to the code, and using a consistent mechanism that is well-defined and structured to enable tooling.

The specifics of this administrative model are not covered within this document.

10.5. Defining Property Constraints

You can set up constraints on control properties using the `@AnnotationConstraints` annotation. `@AnnotationConstraints` allows you to set up rules related to (1) the instantiation of the control properties by client code, (2) external overriding of the control, and (3) the Beehive runtime version required by the control.

Note: the constraint rules are enforced at build time, when controls are declared in client code by `@Control`. There is no runtime enforcement of the rules.

For example the following constraints require that

- all attributes must be referenced when declaring the control BookControl
- the values of the "title" and "subject" attributes must not exceed 10 characters in length
- the value of the "content" attribute must not exceed 200 characters in length

```
import java.lang.annotation.*;

import
org.apache.beehive.controls.api.bean.AnnotationContstraints.MembershipRule;
import
org.apache.beehive.controls.api.bean.AnnotationContstraints.MembershipRuleValues;
import org.apache.beehive.controls.api.properties.PropertySet;

@ControlInterface
public interface BookControl
{
    ...

    /**
     * The user must set all attribute values when
     * instantiating controls declaratively.
     */
    @PropertySet
    @Target ({ElementType.FIELD, ElementType.TYPE})
    @Retention(RetentionPolicy.RUNTIME)
    @AnnotationConstraints.MembershipRule(
     AnnotationConstraints.MembershipRuleValues.ALL_IF_ANY)
    public @interface Intro
    {
        @AnnotationMemberTypes.Text(maxLength=10)
        public String title();
```

```
        @AnnotationMemberTypes.Text(maxLength=10)
        public String subject();
        @AnnotationMemberTypes.Text(maxLength=200)
        public String content();
    }

    ...
}
```

The following client code will cause a compile error, because it violates two of the constraints:

- The "all if any" constraint on the BookControl.Intro annotation is violated because only two (title and subject) of the three attributes (title, subject, and content) are referenced.
- The subject attribute's value exceeds 10 characters in length.

```
@Control
@BookControl.Intro( title="title", subject="subject of the book" )
BookControlBean myBook
```

Not all Java types are supported by `@AnnotationMemberTypes`. For a list of the supported types see Interface AnnotationMemberTypes.

11. Extensibility

The Controls architecture supports an extensibility model that enables the declarations of user-defined operations or events, based upon a predefined set of semantics defined by the author of the Control type. The extensibility mechanism enables the definition of an interface to the resource where operations (or events) have very specific context.

For example, in the JmsMessageControl sample, the extensibility mechanism will be used to raise the level of abstraction: instead of a low-level mechanism to enqueue messages to a topic or queue, the Control enables extensibility where operations can be defined that correspond to enqueuing messages with a very specific format and set of properties, and where message or property content is derived from method parameters. This creates a logical view of the resource (in this case a queue or topic) where the operations available on it have very specific (and constrained) semantics.

For this section, we'll start with the how an extension is defined, look at the authoring model for defining an extensible Control type, and finally show the client view of using an extended type.

11.1. Defining an Extended Interface for a Control Type

An extension to a base Control type that defines a specific resource use case is created by defining a new Control type that derives from the original type and is annotated with the ControlExtension annotation type:

Declaring a Control Extension (Control Extension Interface)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlExtension;

import org.apache.beehive.controls.examples.JmsMessageControl.*;

@ControlExtension
@Destination(type=DestinationType.QUEUE, name="queue.orders")
public interface OrderQueue extends JmsMessageControl
{
    ...
}
```

This example shows how property values can be configured on the extended interface to further parameterize the use case. In this case, the InvoiceQueue interface is being designed for a very specific use case: to enable orders to be enqueued to a JMS queue named "queue.orders".

Once defined, the Control extension author can now begin to define additional operations on it, in this case the ability to enqueue messages to the OrderQueue by calling methods on it.

Declaring Extended Operations with Properties (Control Extension Interface)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlExtension;

@ControlExtension
@Destination(type=JmsMessageControl.QUEUE, name="queue.orders")
public interface OrderQueue extends JmsMessageControl
{
    public class Order implements java.io.Serializable
    {
        public Order(int buyer, String[] list){
            buyerID = buyer;
            itemList  list;
        }
        private int buyerID;
        private String[ ] itemList;
    }

    @Message (MessageType.OBJECT)
    public void submitOrder(
        @Body Order order,
        @Property ( name="DeliverBy" ) String deliverBy
    );
}
```

This interface defines a single operation, submitOrder, that enqueues an ObjectMessage containing a new order. The body of the message will be a single instance of the Order class, and it will have a single StringProperty with the expected delivery date (enabling message

selector-based queries for orders that are past due).

The message format (in this case an ObjectMessage) and the mapping of operation parameters to message content and/or properties are all defined using metadata annotations on the method or its parameters. This format makes it very easy for tools to assist in the creation and presentation of extension interfaces.

How does the extension author (or tool) know about the set of annotations that can be used on the extension interface? This is the topic of the next section.

11.2. Defining Extension Semantics for a Control Type

A Control author is responsible for defining the extensibility semantics for a particular type, since ultimately they are responsible for providing the implementation that fulfills the semantics.

The extension semantics for a Control are part of the public contract for the Control, and thus are defined on the Control Public Interface as well. As with Control properties, these are defined in the form of metadata annotation interfaces, as show in the following sample code from the JmsMessageControl Public Interface:

Declaring Extension Semantics (Control Public Interface)

```
package org.apache.beehive.controls.examples;

import java.io.Serializable;
java.lang.annotation.ElementType
import java.lang.annotations.Retention;
import java.lang.annotations.RetentionPolicy;
import java.lang.annotations.Target;

import org.apache.beehive.controls.api.bean.ControlInterface;

@ControlInterface
public interface JmsMessageControl
{
    ...
    public enum MessageType { BYTES, MAP, OBJECT, STREAM, TEXT }

    @Target({ElementType.METHOD})
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Message
    {
        public MessageType value() default MessageType.TEXT;
    }

    @Target({ElementType.PARAMETER}
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Body {}
```

```
    @Target({ElementType.PARAMETER})
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Property
    {
        public String name();
    }
}
```

The JmsMessageMessageControl defines three annotation types: Message, Body, and Property. The @Target annotation on the Message declaration specifies that Message can be placed on the method declaration to indicate the type of JMS message that will be enqueued by the operation. The Body annotation is used to indicate the method parameter that contains the contents of the message (and must have a type that is compatible with the specified MessageType). The Property annotation on a method parameter indicates that the parameter's value should be stored as a property on the enqueue message, with the property name coming from the value of the annotation and the property type derived from the type of the method parameter.

The key is that the Control Public Interface contains sufficient details about the expected annotations that a tool can support the construction. It also makes it possible for the Control compiler (that converts the extended interface to an associated bean implementation) to perform validation of interface and method annotations.

More details on how these extension semantics are implemented are described in the next section.

11.3. Authoring an Extensible Control Type

The author of a Control type is responsible for providing the code that implements the extension semantics for the Control. Support for extensibility is optional; so a Control author indicates extensibility of a type by declaring that that the Control Implementation Class implements the org.apache.beehive.controls.api.bean.Extensible interface. This interface has a single method named invoke().

The skeleton of this code for the JmsMessageControlImpl class is shown below:

Implementing Extended Operations (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.api.context.Context;
import org.apache.beehive.controls.api.context.ControlBeanContext;
import org.apache.beehive.controls.api.bean.Extensible;
import java.lang.reflect.Method;

@ControlImplementation(isTransient=true)
```

```
public class JmsMessageControlImpl implements JmsMessageControl, Extensible
{
    @Context ControlBeanContext context;

    public Object invoke(Method m, Object [] args) throws Throwable
    {
        //   Extensibility implementation
        ...
    }
}
```

The invoke() method on the Control Implementation Class will be called any time an operation defined on an extension interface is called on the Control by its client. The implementation of this method has responsibility for examining the current set of properties for the Control instance, methods, and parameters and using them to parameterize the behavior of the Control.

This is demonstrated by the code below, which shows a portion of the implementation of invoke() for the JmsMessageControlImpl class:

Accessing Method Properties Using the Context (Control Implementation)

```
Object invoke(Method m, Object [] args) throws Throwable
{
    ...
    int bodyIndex = 1;
    for (int i= 0; i < args.length; i++)
        if (context.getArgumentPropertySet(m, i,
JMMessageControl.Body.class) != null)
          bodyIndex = i;

    //
    // Create a message of the appropriate type
    //
    Message msg = null;
    JMSMessageControl.Message msgProp =
      context.getMethodPropertySet(m, JMSMessageControl.Message.class);
    switch(msgProp.value())
    {
        case MessageType.OBJECT:
            msg = session.createObjectMessage(args[bodyIndex]);
            break;
            ...
    }

    //
    // Decorate the message with properties defined by any arguments
    //
    for (int i= 0; i < args.length; i++)
    {
        JMSMessageControl.Property jmsProp =
          context.getParameterPropertySet(m,i,
JmsMessageControl.Property.class);
```

```
        if (jmsgProp != null)
        {
            String name = jmsProp.value();
            if (args[I] instanceof String)
                msg.setStringProperty(name, ((String)args[i]);
            else if (args[I] instanceof Integer)
                ...
            else
                msg.setObjectProperty(name, args[I);
    }
}
```

In the sample code above, the Control Implementation Class uses the ControlBeanContext getMethodProperty and getParameterProperty APIs to query properties of the invoked method and its argument. These query methods will return null if the property is not found and no default was defined for the attribute member.

11.4. Client Model for Using an Extended Control Type

The client model for using an extended Control type is exactly the same as the model for using a base Control type. The same set of declarative and programmatic instantiation mechanisms (described in the previous section) will be used, and operations or events are handled the same way.

Below is sample code that uses the OrderQueue extended type (using declarative client model):

Using a Control Extension (Client Code)

```
    @Control org.apache.beehive.controls.examples.OrderQueueBean orderBean;
    ...
    Order order = new OrderQueue.Order();
    order.buyerID = myID;
    order.itemList = new String [] {"item1", "item2"};
    orderBean.submitOrder(order, "12-31-2004");
```

Looking closely at the example, you'll notice that a derived ControlBean type (OrderQueueBean) is generated by the Control compiler, just as it is for a base Control type. The skeleton of this ControlBean Generated Class is shown below:

Implementation of Extended Operations (ControlBean Generated Class)

```
package org.apache.beehive.controls.examples;

public class OrderQueueBean
  extends JmsMessageControlBean
  implements OrderQueue
{
    JmsMessageControlImpl _impl;
    ...
    public void submitOrder(Object order, String deliveryBy)
```

```
    {
        ...
        _impl.invoke(submitOrderMethod, new Object [] {order, deliveryBy};
        ...
    }
}
```

There are several attributes worth noting about the extended ControlBean Generated Class:

- Its implementation will be a subclass of the base type ControlBean, so implementation of base type operations is inherited.
- The extended bean will implement the extended Control interface, meaning all extended operations will be implemented by the bean.

The implementation of these extended operations will always delegate down to the base Control Implementation Class by calling the Extensible.invoke() method.

12. Composition

The Controls architecture supports a composition model, based upon the JavaBeans Runtime Containment and Services Protocol. This means that it is possible for new types of ControlBeans to be defined that are built through composition of one or more other types.

12.1. Composition Using Declarative Instantiation

Additionally, the ControlBeans authoring model makes composition very simple based upon the declarative instantiation model. Within any ControlBean implementation, any @Control fields will automatically be initialized as children of the local bean's context.

Here's a simple example based upon our previous OrderQueue example. Let's say that we want to create a logical Control that can be used to submit orders. This Control will submit to one of two different queues, depending upon whether the order needs to ship in less than 30 days, or greater than 30 days.

The implementation of this Control could look like:

Composition Using Declarative Instantiation (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

@ControlImplementation(isTransient=true)
public class OrderRouterImpl implements OrderRouter
{
    @Control @Destination(Name="RushOrders")
    OrderQueueBean rushOrders;

    @Control @Destination(Name="Orders")
    OrderQueueBean orders;
```

```
    ...

    public void submitOrder(Order order, String deliverBy)
    {
        if (needsRushDelivery(deliveryBy))
            rushOrders.submitOrder(order, deliverBy);
        else
            orders.submitOrder(order, deliverBy);
    }
}
```

In this example, the OrderRouterImpl Control itself uses the services of two different OrderQueue Controls referencing two different queues, and uses a helper method (needsRushDelivery) to decide where to enqueue a particular order. The new Control has the same operations exposed as the original Controls; but now uses the services of one or the other of its children to satisfy the request.

The next section describes doing an equivalent composition using mechanisms to instantiate and build the Control hierarchy.

12.1.1. Composition using Programmatic Mechanisms

Because the ControlBeans architecture is built using the JavaBeans Runtime Containment protocol, which defines a base composition model for JavaBeans, it is also possible to manually instantiate and Controls using the APIs it defines. The ControlBeanContext API extends the java.beans.beancontext.BeanContext API, which provides support for adding children to the current bean's context.

Here's the previous sample, rewritten to use programmatic composition:

Composition Using Programmatic Instantiation (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

@ControlImplementation(isTransient=true)
public class OrderRouterImpl implements OrderRouter
{
    // no @Control annotation, so no auto-init
    OrderQueueBean rushOrders;
    // no @Control annotation, so no auto-init
    OrderQueueBean orders;
     @Context ControlBeanContext context;
    ...

    public void context_onCreate()
    {
        ClassLoader cl = Thread.currentThread().getContextClassLoader();
        rushOrders =
          (OrderQueueBean)Beans.instantiate(cl,
```

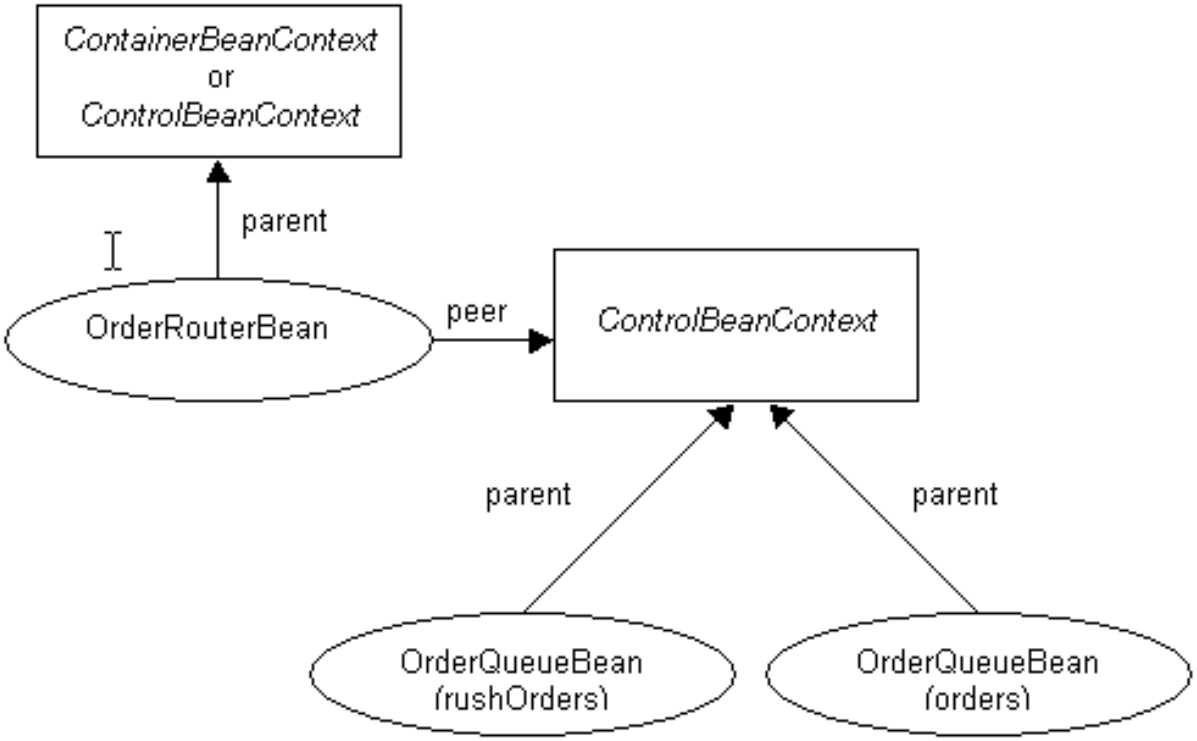
```
            "org.apache.beehive.controls.examples.OrderQueueBean");
        rushOrders.setDestinationName("RushOrders");
        context.add(rushOrders);
        orders =
          (OrderQueueBean)Beans.instantiate(cl,
            "org.apache.beehive.controls.examples.OrderQueueBean");
        orders.setDestinationName("RushOrders");
        context.add(orders);
    }

    public void submitOrder(Order order, String deliverBy)
    {
        ...
    }
}
```

12.2. Internal Architecture for Composition and Services

The JavaBeans Runtime Containment and Services Protocol provides the base composition model for Control composition and containment. In this model, JavaBeans are associated with a BeanContext that manages the composition hierarchy and also manages any contextual services requested by the contained beans.

In the Control architecture, a ControlBean will potentially be related to two different BeanContexts: a parent context that represents the outer container for the bean, and a peer context that provides containment and services to other beans nested within that Control.

These context relationships from the previous sample are shown in the following diagram:



In the diagram, the two OrderQueueBean instances created by OrderRouterBean are nested within the ControlBeanContext; while not shown, these two beans would also have a peer ControlBeanContext providing them with contextual services.

The peer ControlBeanContext provides localized generic services to the associated Control Implementation instance, such as ability to resolve property values from the local bean instance or externalized configuration, and the delivery of lifecycle events. The ControlBean architecture uses a delegation model for service discovery. If an implementation instance requests a service that is not implemented by the peer BeanContext, it will delegate up to the parent context to find a provider for the service.

At the root of the bean composition hierarchy is an instance of a ContainerBeanContext. This context represents the external runtime environment, within which the ControlBean is running. This might represent an EJB, servlet, web service, Java application, or any ControlBean-capable container. The ContainerBeanContext is responsible for the initialization and provisioning of service providers that are specific to runtime environment with which it is associated.

Whether ContainerBeanContext or ControlBeanContext, the BeanContext instances also provide the basic hierarchy of composition, as shown by the parent-child relationships above.

13. Inheritance

The Controls architecture also makes it possible to extend the functionality of existing Controls using standard Java inheritance. While more complex scenarios are possible, a common model for extending a Control type using inheritance involves extending both public interface and the implementation to extend base functionality by adding new operations, events, or properties.

The following code sample shows the basic structure:

Basic Inheritance Sample Code

```
// A.java: The base control interface
@ControlInterface
public interface A { ... }

// AImpl.java: The implementation of the base control interface
@ControlImplementation
public class AImpl implements A { ... }

// B.java: The extension of the base interface that adds
// operations, properties, and/or events
@ControlInterface
public interface B extends A { ... }

// BImpl.java: The implementation of the extended control interface
@ControlImplementation
public class BImpl implements B { ... }
```

In the example above, the BBean JavaBean class that results from processing of B.java will expose the operations, properties, and events defined by both the A and B control interfaces. The BImpl class would need to implement all operations defined by the B interface, and could also choose to override some, all, or none of the operations defined by A.

Inheritance is also supported for extensible control types. If AImpl implements the Extensible interface, then BImpl could choose to define additional extensibility PropertySets and implement a new Extensible.invoke() method to provide their semantics (delegating to AImpl.invoke() as appropriate). It could also choose not to extend the extensibilty semantics and allow all operations defined within a ControlExtension derived from B to be handled by AImpl.invoke().

14. Context and Resource Events

The Controls programming model includes two contextual services that provide a set of supporting life cycle and resource events to assist the author of a Control Implementation. This section describes the events exposed by these services:

14.1. Life Cycle Events

The ControlBeanContext life cycle events provide notification to the associated ControlBean derived class and Control Implementation Class (and potentially other interested listeners) of significant events related to the peer bean instance.

The Control programming model exposes a basic set of lifecycle events to enable the Control to perform efficient initialization and state management. These events are delivered by the peer ControlBeanContext associated with a ControlBean instance. A listener can register to receive these events using the addLifeCycleListener API on ControlBeanContext; the actual LifeCycle event interface itself is defined there as well:

Context Life Cycle Events

```
package org.apache.beehive.controls.api.context;

public interface ControlBeanContext
    extends java.beans.beancontext.BeanContextServices
{
    ...
    @EventSet
    public interface LifeCycle
    {
        public void onCreate();
        public void onPropertyChange(PropertyChangeEvent pce);
        public void onVetoablePropertyChange(PropertyChangeEvent pce)
                    throws PropertyVetoException;
    }

    public void addLifeCycleListener(LifeCycle listener);
    public void removeLifeCycleListener(LifeCycle listener);
}
```

The specific life cycle and resource events are described in the following section:

14.1.1. The onCreate Event

The onCreate event is delivered when the Control Implementation instance associated with the ControlBean has been constructed and all declarative initialization has been completed. This provides an opportunity for the implementation instance to perform any additional initialization required; implementation instances should generally use the onCreate event instead of writing constructor code.

14.1.2. The onPropertyChange Event

The onPropertyChange event is delivered to a registered listener any time a bound property is changed on the ControlBean. This provides an opportunity for the Control Implementation to

change any internal state that might be dependent upon a property value.

14.1.3. The onVetoableChange Event

The onVetoableChange event is delivered to a registered listener any time a constrained property is changed on the ControlBean. This provides an opportunity for the Control Implementation to validate the set value and veto any client-initiated change if necessary (by throwing a VetoException

14.2. Resource Events

The Control programming model exposes a set of resource events to enable the control to manage external resources (connections, sessions, ...) that it needs to provide its services. The model enables resources to be acquired and held for a resource scope that is determined by the container in which the Controls are executing. For example, in the servlet container, the scope might enable resources to be held for the duration of processing a single http request.

```
package org.apache.beehive.controls.api.context;

public interface ResourceContext
{
    ...
    @EventSet
    public interface ResourceEvents
    {
        public void onAcquire();
        public void onRelease();
    }

    public void addResourceEventsListener(ResourceEvents listener);
    public void removeResourceEventsListener(ResourceEvents listener);
}
```

14.2.1. The onAcquire Event

The onAcquire event is delivered to a registered listener the first time a ControlBean operation is invoked within a particular resource scope. It provides an opportunity for the Control Implementation instance (or other related entities, such as a contextual service provider) to acquire any short-term resources (connections, sessions, etc) needed by the ControlBean.

The onAcquire event is guaranteed to be delivered once (and only once) prior to invocation of any operation within a resource scope; it is also guaranteed that a paired onRelease event will be delivered when the resource scope ends.

For more details on resource management, refer to the Control Overview.

14.2.2. The onRelease Event

The onRelease event is the companion event to onAcquire. It is guaranteed to be called once (and only once) on any bean instance that has received an onAcquire event, when its associated resource scope has ended. It acts as the signal that any short-term resources (connections, sessions, etc) acquired by the Control should be released.

14.3. Receiving Life Cycle or Resource Events

For a Control Implementation Class, the model for receiving context life cycle or resource events is consistent with the general client model for event registration and delivery. Both declarative and programmatic mechanisms are supported.

14.3.1. Declarative Access to events

A Control Implementation Class can receive Life Cycle or Resource Events simply by declaring the annotated @Context Context interface and then defining event handlers that follow the <contextFieldName>_<eventName> convention.

The following sample code shows the JmsMessageControl registering to receive onAcquire and onRelease events:

Declarative Handling of Life Cycle Events (Control Implementation Class)

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.api.context.Context;
import org.apache.beehive.controls.api.context.ResourceContext;
import org.apache.beehive.controls.api.events.EventHandler;

@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl
{
    @Context ResourceContext resourceContext;

    @EventHandler(
      field="resourceContext",
      eventSet=ResourceContext.ResourceEvents.class,
      eventName="onAcquire"
    )
    public void onAcquire()
    {
        // Code to acquire JMS connection/session/destination/writers
        ...
    }

    @EventHandler(
```

```
      field="resourceContext",
      eventSet=ResourceContext.ResourceEvents.class,
      eventName="onRelease"
    )
    public void onRelease()
    {
        // Code to release JMS connection/session/destination/writer
        ...
    }
}
```

When using the declarative mechanism, a Control Implementation Class is free to implement only a subset of the events; it is not necessary that it provide a handler for all events.

14.3.2. Programmatic Access to Events

An external entity (such as contextual service provider or even a client) is also able to register for life cycle events on a ControlBean instance as well. This is done by obtaining a reference to the peer ControlBeanContext for the instance using the getControlBeanContext API, and then using the addLifeCycleListener API to register a lifecycle event listener.

This is shown by the following code:

Programmatic Handling of Life Cycle Events (Control Implementation Class)

```
    JmsMessageControlBean myJmsBean = ...;

    ControlBeanContext peerContext = myBean.getControlBeanContext();
    peerContext.addLifeCycleListener(
        new ControlBeanContext.LifeCycle()
        {
            public void onCreate() { ... };
            public void onPropertyChange(PropertyChangeEvent pce) { ... };
            public void onVetoableChange(PropertyChangeEvent pce) { ... };
        }
    );
```

14.4. JavaBean Context Events

The org.apache.beehive.controls.api.context.ControlBeanContext API extends the following standard JavaBean context APIs:

- java.beans.BeanContextChild
- java.beans.BeanContext
- java.beans.BeanContextServices

These APIs provide access to a standard set of JavaBean events that the Control Implementation Class can register an interest in.

14.4.1. PropertyChange Events

The java.beans.BeanContextChild interface provides the addPropertyChangeListener() and addVetoableChangeListener() APIs to register for notification when a property is modified.

14.4.2. Membership Events

The java.beans.BeanContext interface provides the addMembershipChangeListener() API to register for notification whenever a child is added or removed from the BeanContext.

14.4.3. Context Services Events

The java.beans.BeanContextServices interface provides the addBeanContextServicesListener API to register for notification when new contextual services become available or are revoked.

15. Appendix A: The JmsMessageControl Public Interface

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlInterface;
import org.apache.beehive.controls.api.events.EventSet;
import org.apache.beehive.controls.api.properties.PropertySet;

import javax.jms.Session;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * The JmsMessageControl defines a basic Control to enable messages
 * to be enqueued to a JMS queue or topic.   Using Control properties,
 * you can configure the connection, session, and destination attributes
 * that should be used to connect to the JMS provider.   The Control
 * will transparently connect  to the JMS provider and obtain any
 * necessary resources to enqueue the messages.   The Control will
 * also sure that the resources are properly released at the end of the
 * current resource scope associated with the Control's runtime
 * environment.
 *
 * The Control provides a basic set of operations that allow a simple text
 * or object message to be written to the configured destination.   It also
 * provides an extensibility mechanism that allows new operations to be
 * defined by extending this interface.  Extended operations define the
 * enqueueing of message with a specific type
 * (TextMessage, ObjectMessage, ...) where operation parameters can be
```

```
 * mapped to message properties or content.
 */
@ControlInterface
public interface JmsMessageControl
{
    // OPERATIONS

     /**
      * Sends a simple TextMessage to the Control's destination
      * @param text the contents of the TextMessage
      */
     public void sendTextMessage(String text) throws
javax.jms.JMSException;

    /**
     * Sends a simple ObjectMessage to the Control's destination
     * @param object the object to use as the contents of the message
     */
    public void sendObjectMessage(java.io.Serializable object) throws
javax.jms.JMSException;

    // EVENTS

    /**
     * The Callback interface defines the events for the JmsMessageControl.
     */
    @EventSet
    public interface Callback
    {
        /**
         * The onMessage event is delivered to a registered
         * client listener whenever a
         * message has been sent by the Control.
         * @param msg the message that was sent
         */
        public void onMessage(javax.jms.Message msg);
    }

     // PROPERTIES

    /**
     * The Connection property defines the attributes of the connection
     * and session used to enqueue the message.   This annotation
     * can appear on both class and Control type declarations.
     */
    @PropertySet
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.FIELD, ElementType.TYPE})
    public @interface Connection
    {
        public String factoryName();
        public boolean transacted() default true;
        public int acknowledgeMode() default Session.CLIENT_ACKNOWLEDGE;
    }
```

```
    /** An enumeration that defines the value set of destination types */
    public enum DestinationType { QUEUE, TOPIC }

    /**
     * The Destination property defines the attributes
     * of the JMS destination that should
     * be the target of any enqueued messages.
     */
    @PropertySet
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.FIELD, ElementType.TYPE})
    public @interface Destination
    {
        public DestinationType type() default DestinationType.QUEUE;
        public String name();
    }

    // EXTENSIBILITY ATTRIBUTES

    /**
     * The set of supported message types for extended operations
     */
    public enum MessageType { TEXT, OBJECT, BYTES }

    /**
     * The Message attribute can be placed on an
     * extended operation to describe the format
     * of the message that  should be enqueued when
     * the operation is invoked.  The method is
     * expected to have a least parameter annotated
     * with the Body attribute, and zero or more
     * parameters with the Property attribute
     * defining message properties.
     */
    @Target({ElementType.METHOD})
    public @interface Message
    {
        public MessageType value() default MessageType.TEXT;
    }

    /**
     * The Body attribute indicates that the associated
     * method parameter on an extended operation
     * contains the message body.
     */
    @Target({ElementType.PARAMETER})
    public @interface Body {}

    /**
     * The Property attribute can be used to define
     * operation parameters that should be used to
     * set properties on the message.  The type of
     * property to set will be inferred based upon
```

```
     * the type of the parameter.
     */
    @Target({ElementType.PARAMETER})
    public @interface Property
    {
        public String name();
    }
}
```

16. Appendix B: The JmsMessageControl Implementation Class

```
package org.apache.beehive.controls.examples;

import org.apache.beehive.controls.api.bean.ControlImplementation;
import org.apache.beehive.controls.api.bean.Extensible;
import org.apache.beehive.controls.api.context.Context;
import org.apache.beehive.controls.api.context.ControlBeanContext;
import org.apache.beehive.controls.api.context.ResourceContext;
import org.apache.beehive.controls.api.ControlException;
import org.apache.beehive.controls.api.events.Client;
import org.apache.beehive.controls.api.events.EventHandler;

import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.QueueSender;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSession;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.lang.reflect.Method;

/**
 * The JmsMessageControlImpl class is the
 * Control Implementation Class for the JmsMessageControl.
 * It implements two basic operations
 * (sendTextMessage and sendObjectMessage)
 * as well as an extensibility model that enables custom
 * message formats to be defined and associated with
 * extended method signatures.
 */
@ControlImplementation(isTransient=true)
public class JmsMessageControlImpl implements JmsMessageControl, Extensible
{
    /**
     * The peer BeanContext instance associated with the Control
     */
    @Context ControlBeanContext context;
```

```
    /**
     * The client callback event router for this Control
     */
    @Client Callback client;

    /**
     * The fields are used to hold transient JMS resources
     * that are acquired and held for
     * the resource scope associated with the Control
     */
    transient javax.jms.Connection _connection;
    transient javax.jms.Session _session;
    transient javax.jms.MessageProducer _producer;
    transient javax.jms.Destination _dest;

    /**
     * The Resourceontext instance associated with the Control
     */
    @Context ResourceContext resourceContext;

    /*
     * The onAcquire event handler
     * This method will be called prior to any operation with
     * a given resource scope.  It is responsible for
     * obtaining the connection, session, destination, and appropriate
     * writer instance, for use within the operation.
     */
    @EventHandler(
      field="resourceContext",
      eventSet=ResourceContext.ResourceEvents.class,
      eventName="onAcquire"
    )
    public void  onBeanAcquire()
    {
        //
        // Acquire the property values needed for initialization
        //
        Destination destProp =
          (Destination)context.getControlPropertySet(Destination.class);
        Connection connProp =
          (Connection)context.getControlPropertySet(Connection.class);

        try
        {
            //
            // Obtain the JMS Destination instance based upon the
Destination property
            //
            InitialContext jndiContext = new InitialContext();
            _dest =
(javax.jms.Destination)jndiContext.lookup(destProp.name());

            //
```

```
            // Obtain Connection, Session, and MessageProducer resources
based upon the
            // destination type and the values in the Connection
PropertySet
            //
            if (destProp.type() == JmsMessageControl.DestinationType.QUEUE)
            {
                javax.jms.QueueConnectionFactory connFactory =
(QueueConnectionFactory)jndiContext.lookup(connProp.factoryName());
                _connection = connFactory.createQueueConnection();
                _session =
((QueueConnection)_connection).createQueueSession(
connProp.transacted(),
connProp.acknowledgeMode());
                _producer =
((QueueSession)_session).createSender((Queue)_dest);
            }
            else
            {
                javax.jms.TopicConnectionFactory connFactory =
(TopicConnectionFactory)jndiContext.lookup(connProp.factoryName());
                _connection = connFactory.createTopicConnection();
                _session =
((TopicConnection)_connection).createTopicSession(
connProp.transacted(),
connProp.acknowledgeMode());
                _producer =
((TopicSession)_session).createPublisher((Topic)_dest);

            }
        }
        catch (javax.naming.NamingException ne)
        {
            throw new ControlException("Unable to locate JNDI object", ne);
        }
        catch (ClassCastException ce)
        {
            throw new ControlException("JNDI object did not match expected
type", ce);
        }
        catch (JMSException jmse)
        {
            throw new ControlException("Unable to acquire JMS resources",
jmse);
        }
    }

    /*
     * The onRelease event handler for the associated context
     * This method will release all resource acquired by onAcquire.
     */
    @EventHandler (
      field="resourceContext",
      eventSet=ResourceContext.ResourceEvents.class,
```

```
      eventName="onRelease"
    )
    public void onRelease()
    {
        try
        {
            if (_producer != null)
            {
                _producer.close();
                _producer = null;
            }
            if (_session != null)
            {
                _session.close();
                _session = null;
            }
            if (_connection != null)
            {
                _connection.close();
                _connection = null;
            }
        }
        catch (JMSException jmse)
        {
            throw new ControlException("Unable to release JMS resource",
jmse);
        }
    }

    /**
     * Helper method used to send a message once constructed
     */
    private void sendMessage(javax.jms.Message msg) throws JMSException
    {
        client.onMessage(msg);
        if (_producer instanceof javax.jms.QueueSender)
            ((QueueSender)_producer).send(msg);
        else
            ((TopicPublisher)_producer).publish(msg);
    }


    /**
     * Sends a simple TextMessage to the Control's destination
     * @param text the contents of the TextMessage
     */
    public void sendTextMessage(String text) throws JMSException
    {
        javax.jms.TextMessage msg = _session.createTextMessage(text);
        sendMessage(msg);
    }

    /**
     * Sends a simple ObjectMessage to the Control's destination
```

```
     * @param object the object to use as the contents of the message
     */
    public void sendObjectMessage(java.io.Serializable object) throws
JMSException
    {
        javax.jms.ObjectMessage msg = _session.createObjectMessage(object);
        sendMessage(msg);
    }

    /**
     * Implements the Extensible.invoke() interface for this Control
     * This method uses the Message property to determine the type
     * of message to construct, and then uses the Body and Property
     * attributes of method parameters to supply message
     * content and properties.
     */
    public Object invoke(Method m, Object [] args) throws Throwable
    {
        int bodyIndex = -1;
        for (int i= 0; i < args.length; i++)
        {
            if (context.getParameterPropertySet(m, i,
JmsMessageControl.Body.class) != null)
            {
                bodyIndex = i;
                break;
            }
        }
        if (bodyIndex == -1)
            throw new ControlException(
              "No @Body argument defined for operation: "
              + m.getName()
            );

        //
        // Create a message based upon the value of the Message property of
the method
        //
        javax.jms.Message msg = null;
        Message  msgProp = context.getMethodPropertySet(m,
JmsMessageControl.Message.class);
        try
        {
            switch(msgProp.value())
            {
                case TEXT:
                    msg =
_session.createTextMessage((String)args[bodyIndex]);
                    break;

                case OBJECT:
                    msg =
_session.createObjectMessage((java.io.Serializable)args[bodyIndex]);
                    break;
```

```
                case BYTES:
                    javax.jms.BytesMessage bmsg;
                    msg = bmsg = _session.createBytesMessage();
                    bmsg.writeBytes((byte []) args[bodyIndex]);
                    break;
            }
        }
        catch (ClassCastException cce)
        {
            throw new ControlException("Invalid type for Body parameter",
cce);
        }

        //
        // Now decorate the message with any Property-annotated parameters
        //
        for (int i= 0; i < args.length; i++)
        {
            JmsMessageControl.Property prop =
                context.getParameterPropertySet(m, i,
JmsMessageControl.Property.class);
            if (prop != null)
            {
                String propName = prop.name();
                if (args[i] instanceof String)
                    msg.setStringProperty(propName, (String)args[i]);
                else if (args[i] instanceof Integer)
                    msg.setIntProperty(propName,
((Integer)args[i]).intValue());
                else if (args[i] instanceof Short)
                    msg.setShortProperty(propName,
((Short)args[i]).shortValue());
                else if (args[i] instanceof Boolean)
                    msg.setBooleanProperty(propName,
((Boolean)args[i]).booleanValue());
                else if (args[i] instanceof Float)
                    msg.setFloatProperty(propName,
((Float)args[i]).floatValue());
                else if (args[i] instanceof Double)
                    msg.setDoubleProperty(propName,
((Double)args[i]).doubleValue());
                else
                    msg.setObjectProperty(propName, args[i]);
            }
        }

        //
        // Send it
        //
        sendMessage(msg);
        return msg;
    }
}
```