Controls AnnotationBasedFeatures

Implementing Annotation-based Controls Programming Model Features Using Interceptors

Many interesting controls programming model features are annotation-based; the presence and values of annotations on control interfaces and extensions drive significant runtime behaviour differences. Examples of such features include async control operations via message buffering and a variety of security features (run-as, role authorization enforcement, etc).

This proposal describes a general interceptor mechanism for implementing such annotation-based features.

Summary

Provide the ability to associate a J'*_avaBeans service interface with an annotation to define its runtime feature behaviour. These interfaces are treated as "interceptors" in the controls runtime, which will automatically instantiate and execute implementations of them at the appropriate execution points (pre /post invocation of a control operation, etc).

Feature implementors are required to:

- Define a unique J_*'avaBeans service interface for the desired feature. It must extend the `InterceptorContext` interface, which defines the contract that the controls runtime has with interceptors.
- Define the annotation(s) that control the desired feature. Those annotations are meta-annotated with the `@InterceptorAnnotation`, which indicates to the infrastructure that this is a controls-related annotation, and specifies the J'*_avaBeans service interface (declared above) that maps to implementation(s) of the feature.
- Provide an apt annotation processor that at minimum implements the check() method of the `TwoPhaseAnnotationProcessor` abstract class and enforces the compile-time semantics of the new annotation(s). This annotation processor must be configured to be discoverable by apt during controls compilation.
- Provide at least one implementation of their J_*'avaBeans service interface, typically using features specific to a particular controls container environment.
- Register (via addService()) their service implementation in the appropriate control container. The base container `ControlContainerContext` may register some implementations of some of these features already; downstream container providers (for WLS, Tomcat, Geronimo, etc), may override and/or provide additional services.

Clients of the feature may then:

- Annotate control types (interfaces and extensions) with the newly defined annotations.

The controls compilation process does the following:

- Runs interceptor-related annotation processors, producing the appropriate diagnostics with respect to usage of interceptor-related annotations.
- Analyzes controls and code-gens (into the generated `ControlBean` class) implicit references to any interceptor services required to support the features those controls use, initialization code for those implicit references, and calls to the services based on the location(s) of the annotation(s) in the controls. For example, if the interceptor-based annotation is annotating a control method, then there would be a code-generated call to that interceptor pre and post invocation of the actual impl method.

The controls runtime:

- Provides support code (mostly in the `ControlBean` base class) to help manage interceptor lifecycle,

persistence (ie, ensuring that the aren't serialized, but are re-init'ed on load), interceptor priority/ordering, and actual interceptor execution.

Details

For the purposes of a detailed example, we'll define an annotation called `@MessageBuffer` and show how to associate it with a control interceptor to provide asynchronous buffering of control operation invocation.

`InterceptorContext` – the interface that interceptor services must extend:

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

public interface InterceptorContext
{
    /* NOTE: Could have InterceptorContext extend InvokeListener to pick up preInvoke/postInvoke,
             but the signatures on InvokeListener (rightly?) exclude ControlBeanContext */

    /** Called before a control operation is invoked */
    public preInvoke( Method m, Object [] args, ControlBeanContext cbc );
    /** Called after a control operation is invoked */
    public postInvoke( Method m, Object [] args, ControlBeanContext cbc );

    /** Called before a control event is fired (through a client proxy) */
    public preEvent( Class eventSet, Method m, Object [] args, ControlBeanContext cbc );
    /** Called after a control event is fired (through a client proxy) */
    public postEvent( Class eventSet, Method m, Object [] args, ControlBeanContext cbc );

    /** Called when a control impl instance is created */
    public onControlCreate( ControlBeanContext cbc );
    /** Called before a control (bean?) is persisted */
    public onControlSerialize( ControlBeanContext cbc );
    /** Called after a control (bean?) is read from persistent storage */
    public onControlDeserialize( ControlBeanContext cbc );
}
```

TBD: allow interceptor services to extend context eventsets? For example, this would allow an interceptor service to extend `ControlBeanContext.LifeCycle` or `ResourceContext.ResourceEvents` and the controls framework would auto-register the impls as the appropriate listener(s). This could get confusing though.. and what is the limitation on which eventsets behave in this manner?

The `MessageBuffer` interceptor service interface

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

public interface MessageBufferContext extends InterceptorContext
{
}
```

A particular implementation of the `MessageBuffer` interceptor service.

```
public class MessageBufferImpl implements MessageBufferContext
{
    private static MessageBufferProvider
    {
        //.. boilerplate
    }

    public preInvoke( Method m, Object [] args, ControlBeanContext cbc )
    {
        // Thread-local check for whether we are being invoked on initial call (in which case
        // we enqueue) or on the post-dequeue call (in which case we do nothing)
        if ( isBufferedCall() )
            return;

        ControlHandle handle = cbc.getControlHandle();

        // include security/auth info?  Call should execute in the right security context on dequeue
        QueueMessage msg = new QueueMessage( handle, m.getName(), args );

        Queue q = getAsyncQueue(); // implemented via JMS, simple in memory queuing system.. ?
        q.enqueue( msg );
    }
}

//
// TODO: a listener on the async queue which will use the control handle
// to dispatch back into the control.  Sets a thread local so the interceptor won't enqueue again.
//
```

`InterceptorAnnotation` – meta-annotation used to identify annotations that are interceptor-based, and bind an interceptor service interface to those annotations:

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

@Documented
@RetentionPolicy(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface InterceptorAnnotation
{
    Class<? extends InterceptorContext> service();
}
```

The `MessageBuffer` annotation that will be used by control authors:

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

import org.apache.beehive.controls.api.context.MessageBufferContext;

@InterceptorAnnotation( service=MessageBufferContext.class )
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MessageBuffer
{
    public boolean value() default true;
}
```

The annotation processor that will enforce the semantics of `MessageBuffer`:

```
public class MessageBufferAnnotationProcessor extends TwoPhaseAnnotationProcessor
{
    // ...

    public void check( Declaration decl )
    {
        // MessageBuffer allowed only on methods with void return type
        if ( decl instanceof MethodDeclaration )
        {
            MethodDeclaration methodDecl = (MethodDeclaraton) decl;
            if ( !(methodDecl.getReturnType() instanceof VoidType) )
                printError( "Message buffered methods must return void" );
        }

        // ... more semantic checks
    }
}
```

A client (.jcx in this case) that uses `@MessageBuffer`.

```
@ControlExtension
public interface MyWebService extends ServiceControl
{
    @MessageBuffer
    public void myWebOperation( int a );
}
```

Content generated into the C'_*ontrolBean:

```
public class MyWebServiceBean extends ControlBean
{
    //
    // For each control operation/event, code-gen a list of applicable interceptors
    // as determined by the annotations on/applying to that operation.
    //
    // Use String instead of Class for late binding.
    //

    static
    {
        private String [] _myWebOperationInterceptors =
            { "org.apache.beehive.controls.api.context.MessageBufferContext" };

        // Allow the container to impose a priority order on interceptor execution.
        _myWebOperationInterceptors = ControlBeanContext.prioritizeInterceptors( _myWebOperationInterceptors );

        // ... additional entries for each operation
    }

    //
    // Each control operation/event already has a method that wraps invocation of the impl method.
    // For interceptors on pre/post invoke, there's no additional codegen here required, the base class
    // impl of pre/postInvoke() do the work and we just pass the list of applicable interceptors.
    //

    public void myWebOperation( int a )
    {
        // ...

        // Pass list of interceptors here
        preInvoke(_myWebOperationMethod, _argArray, _myWebOperationInterceptors);
        try
        {
            _target.myWebOperation(a);
        }
        catch (Throwable t)
        {
            //
            // All exceptions are caught here, so postInvoke processing has visibility into
            // the exception status.  Errors, RuntimExceptions, or declared checked exceptions will
            // be rethrown.
            //
            _thrown = t;

            if (t instanceof Error) throw (Error)t;
            else if (t instanceof RuntimeException) throw (RuntimeException)t;

            throw new UndeclaredThrowableException(t);
        }
        finally
        {
            // Pass list of interceptors here
            postInvoke(_retval, _thrown, _myWebOperationInterceptors);
        }
        return;
    }

    // ...
}
```

Base top-level container service registration:

```
public class ControlContainerContext extends ControlBeanContext
{
    //...

    public initialize()
    {
        super.initialize();

        // Registers a generic, low QoS impl?  Or not at all..
        addService( MessageBufferContext.class, MessageBufferContextProvider.getProvider() );
    }
}
```

Downstream controls container service registration/override:

```
public class MyJ2EEServerControlContainerContext extends ControlContainerContext
{
    public initialize()
    {
        super.initialize();

        // If an existing provider has been registered by a base class, this overrides it
        addService( MessageBufferContext.class, PowerfulMessageBufferContextImpl.getProvider() );
    }
}
```

Runtime support for management and invocation of interceptors is mostly provided in the `ControlBean` class:

```
public class ControlBean
{
    // ...

    //
    // HashMap to hold interceptor impl instances.
    // Populated lazily.  Maps interceptor interface name to impl.
    //

    private final HashMap<String,InterceptorContext> _interceptors = new HashMap<String,InterceptorContext>();

    // ...

    //
    // Retrieves interceptor instances, creates them lazily.
    //

    private InterceptorContext ensureInterceptor( String n )
    {
        InterceptorContext i = _interceptors.get( n );
        if ( i == null )
        {
            i  = getService( getClassLoader().loadClass( n ) );
            _interceptors.put( n, i );
        }
        return i;
    }

    // ...

    protected void preInvoke( Method m, Object [] args, String [] interceptorNames )
    {
        //...

        ControlBeanContext cbc = getControlBeanContext();

        for ( String n : interceptorNames )
        {
            InterceptorContext i = ensureInterceptor( n );
            i.preInvoke( m, args, cbc );
        }

        //..
    }

    // ... similarly for postInvoke, etc
}
```

Ordering of interceptors is configured by `ControlBeanContext`:

```
public class ControlBeanContext
{
    public static String[] prioritizeInterceptors( String [] interceptorNames )
    {
        // obtain prioritization from external configuration (load a resource?)
        // or hardcoding.  Process incoming interceptorNames array and return
        // an array in the desired order.
    }
}
```

*TBD:' *it may make sense to make this support non-static, and do it in the init/constructor for C*'_'ontrolBeans – this would make it easier to delegate to downstream containers and thus make it easier for the ordering to be container-specific, at the cost of having to do more bookkeeping (instance level instead of static).

Open Issues

- Interceptor contract for events & event handlers?

- Pivot & error semantics for interceptors?
- How does container determine interceptor order? Config file? Config class?
- Signature of `InterceptorContext` methods not deeply thought out
- ...