Collaborative Infrastructure Spring Actors

Author: Ron van Hoof
Version: 1.0 12/17/2009

Table of Contents

1 Introduction 3
2 CI Spring Actors Application 3
3 Actors 5
3.1 GUI Actor 5
3.1.1 IGuiActor Interface 5
3.1.2 GuiActor Class 5
3.1.3 GuiActorProxy Class 7
3.1.4 CI Configuration 8
3.1.5 Spring Configuration 11
3.2 KFX Parser Actor 13
3.2.1 IKfxParserActor Interface 13
3.2.2 KfxParserActor Class 14
3.2.3 KfxParserActorProxy Class 15
3.2.4 CI Configuration 15
3.2.5 Spring Configuration 18
4 Data Services 20
4.1 KFX File Data Representation 21
4.2 Data Store 24
4.3 Object Relational Mapping with Hibernate 26
4.4 KFX File Data Access Object 33
4.4.1 KFX File DAO Interface 33
4.4.2 Hibernate Session Factory 35
4.4.3 Hibernate KFX File DAO Class 36
4.4.4 DAO Spring Configuration 37
4.5 KFX File Service 38
4.5.1 IKfxFileService Interface 38

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

4.5.2 KfxFileService Class 39
4.5.3 Spring Configuration 40
4.6 Transactions 40
4.6.1 Transaction Manager 41
4.6.2 Transaction Attributes 42
4.6.3 Declaring Transactions 45
5 Application Services 47
5.1 User Interface Application Service 47
5.1.1 IUIApplication Interface 47
5.1.2 UIApplication Class 47
5.1.3 Spring Configuration 53
5.2 KFX Parser Service 54
5.2.1 IKfxParser Interface 54
5.2.2 KfxParser Class 54
5.2.3 Spring Configuration 55
5.3 CI Message Handling 56
5.3.1 CIMessageHandlerFactory Class 56
5.3.2 ICIActionMessageHandler Interface 58
5.3.3 ParseMessageHandler Class 58
5.3.4 Spring Configuration 62
6 Packaging 63
6.1 Prerequisites 63
6.2 File Organization 63
6.3 Configuring the Actor Hosting Environments 66
6.4 Actor Hosting Environment Startup Script 69
6.5 Spring Selector Configuration File 70
7 Running the Application 71
8 Special Notes 72
8.1 AOP.xml 72
9 References 73

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

1 Introduction

The Collaborative Infrastructure (CI) provides a communication infrastructure to allow components (actors) to easily collaborate with one another as part of a distributed system in a structured agent-oriented fashion. With the integration of the Spring application framework with the CI it is now possible to allow an actor developer to focus the development of a CI-based enterprise application on the development of the business logic using POJOs (Plain Old Java Objects). The Spring application framework manages most of the plumbing for integrating the business objects and for having these business objects interact with enterprise technologies such as object relational mapping, database interactions, transactions, and security. For details about the integration between the CI and Spring see the document 'Integrating Spring with the Collaborative Infrastructure'[1].

This document provides an example of a small application that utilizes the capabilities of both the Collaborative Infrastructure and Spring, the CI Spring Actors Application.

2 CI Spring Actors Application

The CI Spring Actors application is a distributed application to highlight the capabilities of the integration of the CI and Spring. Figure 1 gives an overview of the application's architecture. This architecture diagram shows actors being hosted in separate actor hosting environments (AHE) with CI services (transport) provided by CI's Actor Service Provider (CI). The other components are POJO's managed by Spring. The black dots with lines indicate Spring based dependency injection where the relations between the objects are created by Spring via a configuration file. The wiring of these objects is done at run-time not at compile time.

This application does the following and is a subset of and loosely based on an application developed for NASA called OCAMS (Orbital Communications Adapter Management System). There are two actors, the GUI Actor and KFX Parser Actor. The GUI Actor manages the display of the user interface to the user via the UIApplication. The KFX Parser Actor provides KFX (Ku-band File Transfer) log parsing services. Parsed KFX File objects are stored in a MySQL database. These KFX File objects are mapped to a relational database via an Object Relational Mapping (ORM). The object relational mapping is managed by Hibernate in this example.

When the GUI actor is started it displays the user interface (UI). The UI has a parse button and a list view. The parse button is not enabled until the GUI application has found the KFX Parser Actor via the CI's directory service. When enabled and pressing the parse button a Communicative Act (message) is sent to the KFX Parser actor via the CI's transport service. Upon receipt the KFX Parser actor reviews the payload of that Communicative Act and uses a CI MessageHandler Factory to determine if it has an appropriate message handler registered for it. If so the factory will hand off the Communicative Act to that message handler for processing. In this case the Parse Request Handler. This message handler processes the message, has the KFX Parser 'parse the logs' and store the parsed data into the database via the KFX File Service.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

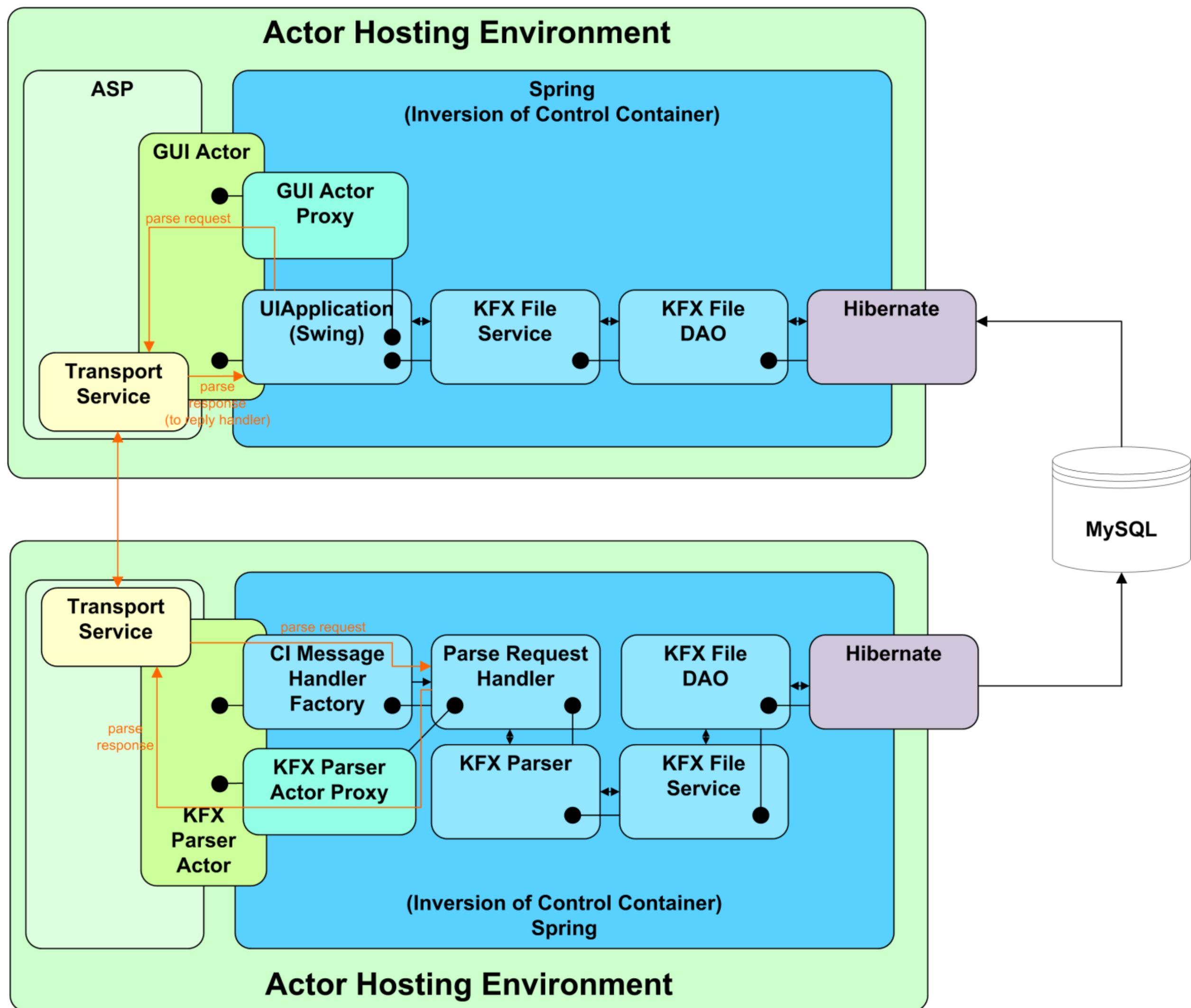


Figure 1: CI Spring Actors Application Architecture

The KFX File Service uses the KFX File Data Access Object to have the file objects stored via Hibernate into the MySQL database. The parser actor then creates a response Communicative Act with a payload containing the keys uniquely identifying the KFX files. This response Communicative Act is sent via the CI's transport service to the GUI Actor.

The GUI Actor registered a CI Reply Handler to process responses for its request. The CI Transport Service on receipt of a response hands the Communicative Act over to that reply handler. As part of the response processing the UI application retrieves the KFX Files from the database using the provided keys and displays the keys in the list.

The transaction boundaries for the database transactions are declared declaratively in the Spring configuration file as is the data source used for storing the data (Hibernate/MySQL). With this approach it is easy to change the transaction manager, ORM tool, and database without having to change the application code (except for the DAO classes which are tied to the ORM tool). Hibernate abstracts away

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

the database so we can swap in any database we want and just change the appropriate configuration files.

3 Actors

The application consists of two actors, the GUI Actor and KFX Parser Actor. Both are CI actors and need to be made available as Spring beans to support dependency injection. Each actor follows the steps for developing a 'Springified' actor as outlined in the document 'Integrating Spring with the Collaborative Infrastructure'[1]. The following sections will go over the details for each actor, the Java code, the necessary CI actor configuration files, and the necessary Spring configurations.

3.1 GUI Actor

The GUI Actors main responsibility is to display the user interface to the user when the actor is started. In order for the actor to control the visualization it will need a reference to the component managing the visualization. In our example this is the component implementing the IUIApplication interface. The interface and implementation of this component is discussed later in this document.

To develop the GUI Actor and to allow us to make it an injectable bean and to allow the actor to be injected into other beans we need to declare an interface for the actor, develop the implementation, and develop a proxy for the actor.

3.1.1 IGuiActor Interface

The IGuiActor interface specifies the contract for the actor to the components that wish to use its services. The contract indicates that it depends on the IUIApplication component for controlling the user interface and that it provides all standard actor services as defined in the ICIActor interface.

```
package gov.nasa.arc.ci.spring.actor.gui;

import gov.nasa.arc.ci.spring.ui.IUIApplication;
import gov.nasa.ci.common.actor.ICIActor;

public interface IGuiActor extends ICIActor {
       public void setApplication(IUIApplication app);
} // IGuiActor
```

By using the interface declaration we hide the implementation details from any components that wish to interact with the actor allowing us to easily proxy the actor which is of important to allow us to make the actor injectable into Spring managed beans.

3.1.2 GuiActor Class

The GuiActor class provides the implementation of the actor.

The construction of CI actors is managed by the Actor Hosting Environment and not by Spring. As a result we cannot just declare the actor as a regular bean in a Spring configuration file. To allow for the configurability of the actor using Spring two things are required, one the bean declaration for the actor must be declared to be abstract, and two the actor implementation/class must be annotated using the

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

Spring Configurable annotation. Spring uses an AspectJ aspect to initiate configuration of the actor when the actor is instantiated by the Actor Hosting Environment. As soon as the actor is created, before the actor is initialized, Spring will inject all beans and values as configured in the Spring configuration file. Note that the identifier assigned to the abstract actor bean as declared in the Spring configuration file must be provided as an argument to the Configurable annotation. The combination of the abstract bean and the Configurable annotation makes the CI actor Spring configurable.

The problem with abstract beans is that they cannot be injected into other Spring beans. To allow for the CI actor to be injected into Spring configured beans a proxy to the actor is required where the proxy is a Spring bean. The proxy implementation is covered in the next section. Since we cannot inject a reference to this actor into the proxy we need to have the actor inject itself into the proxy somehow. This can be accomplished by configuring Spring to inject the proxy into the actor and to as part of the implementation have the actor set a reference to itself in the proxy. The proxy can then delegate all invocations intended for the actor to the actor it is a proxy for. The support for making a proxy injectable into the actor has been made available in the class AbstractSpringCIActor. Our actor therefore extends this class.

To ensure that the actor abides by its contract it implements the IGuiActor interface.

Most of the actor's CI related implementation is covered by AbstractCIActor which is extended by the AbstractSpringCIActor. The use of this class minimizes the amount of code that needs to be written to develop an actor. The CI requires a subset of methods to be implemented by the actor developer. These methods are related to actor control and management. Methods related to pausing, resuming, and resetting an actor can optionally be implemented. See the Javadoc documentation for the signature of those methods if you wish to implement them.

```
package gov.nasa.arc.ci.spring.actor.gui;

import javax.swing.SwingUtilities;

import gov.nasa.arc.ci.spring.ui.IUIApplication;
import gov.nasa.ci.corba.api.ActorException;
import gov.nasa.ci.spring.actor.AbstractSpringCIActor;

import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Configurable;

@Configurable("guiActorImpl")
public class GuiActor
      extends AbstractSpringCIActor
      implements IGuiActor {

  private final static Logger LOGGER = Logger.getLogger(GuiActor.class);

  /** The application that serves as the interface with the user */
  private IUIApplication m_oApplication;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    public void setApplication(IUIApplication app) {
      m_oApplication = app;
    } // setApplication

      //
      // ***** Actor Control and Management *****
      //

    public void onInitialize() throws ActorException {
      m_oApplication.initialize();
    } // onInitialize

    public void onStart() throws ActorException {
      // visualize the application
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          m_oApplication.setVisible(true);
        } // run
      });
    } // onStart

    public void onStop() throws ActorException {
    } // onStop

    public void onShutdown() {
    } // onShutdown
  } // GuiActor
```

In the implementation of the actor we take care of implementing support for injecting a reference to an IUIApplication (`setApplication`), manage the initialization of the user interface in `onInitialize` by invoking the initialize method on the injected IUIApplication reference, and initiate the display of the user interface when the actor is started in `onStart`. Note that we make the application visible on the AWT event queue using SwingUtilities.invokeLater. As recommended for Swing all UI operations should take place on this event queue. The Actor Hosting Environment uses its own thread of execution to start the actor, which is not the AWT event queue.

3.1.3 GuiActorProxy Class

As mentioned before in order for an actor to be injectable in other beans we need to use an actor proxy. The proxy is a Spring managed bean. When an actor needs to be injected into a Spring managed bean it is required that the proxy is injected into those beans, and not the actor itself. Spring would raise exceptions if you try to inject the actor bean into a Spring managed bean indicating that Spring cannot instantiate abstract beans.

Developing this proxy is pretty painless. The proxy needs to indicate that it complies with the actor's contract by implementing the IGuiActor interface. The CI provides the CIActorProxy base class which provides the implementation for all CI related methods. The actor's proxy then only needs to implement the business methods declared in the actor's interface, in our case `setApplication`. All the implementation of this method needs to do is delegate the call to the proxied actor. The CIActorProxy class provides the method `ICIActor getActor()` to get access to the provided actor.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
package gov.nasa.arc.ci.spring.actor.gui;

import gov.nasa.arc.ci.spring.ui.IUIApplication;
import gov.nasa.ci.common.actor.CIActorProxy;

public class GuiActorProxy extends CIActorProxy implements IGuiActor {
  public void setApplication(IUIApplication app) {
    ((IGuiActor)getActor()).setApplication(app);
  } // setApplication
} // GuiActorProxy
```

3.1.4 CI Configuration

The coding of the interface, class, and proxy completes the Java implementation for the actor. To allow the CI Actor Hosting Environment to load and properly configure the actor an actor descriptor must be created. This actor descriptor specifies the name of the Java Class for the actor that needs to be instantiated to create the actor, the simple and qualified names to be assigned to the actor, indicates whether and if so with what names to register the actor in the directory service, optionally defines its capabilities, any service related properties, and actor specific properties.

The following is the actor configuration file for the GUI Actor. The file is named GuiActor.ciax. In general for actors only the following elements need to be update:

LIBRARY - To specify the class name for your actor class
SIMPLENAME - The simple name assigned to your actor (mainly for logging purposes)
QUALIFIEDNAME - The globally unique name assigned to your actor
DIRECTORYNAMES - The directory names by which the actor is registered in the directory service

Optionally you can define the configuration file that needs to be loaded for the actor with some actor specific configurations by defining the property `ci.actor.file.configuration`.

To enable the publication of a heart beat for the actor add the property ci.actor.heartbeat.interval with as a value the interval in milliseconds. Actors can subscribe for actor heart beats with the management service and use this to discover actors. Without this property no heart beat will be published.

The CI includes a template for this actor descriptor file. The descriptor file should be placed in the deployment directory configured for the Actor Hosting Environment.

```
<?xml version="1.0" encoding="UTF-8"?>

<ACTOR>
  <!-- The library in which the actor's implementation can be found
       The library element value is the name of the Java
       class that provides the actor implementation.
  -->
  <LIBRARY type="ci.actor.class">gov.nasa.arc.ci.spring.actor.gui.GuiActor</LIBRARY>

  <!-- The credentials of the actor -->
  <CREDENTIALS>
    <!-- The simple name for the actor, used for display or logging
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
      purposes -->
  <SIMPLENAME>GuiActor</SIMPLENAME>

  <!-- The globally unique fully qualified name of the actor -->
  <QUALIFIEDNAME>gov.nasa.arc.ci.spring.actor.GuiActor</QUALIFIEDNAME>

  <!-- The directory name(s) with which the actor is
       registered in the directory service. -->
  <DIRECTORYNAMES>
    <DIRECTORYNAME>gov.nasa.arc.ci.spring.actor.GuiActor</DIRECTORYNAME>
    <DIRECTORYNAME>ci/spring/actor/GuiActor</DIRECTORYNAME>
  </DIRECTORYNAMES>

  <!-- Other credentials properties -->
  <PROPERTIES>
  </PROPERTIES>
</CREDENTIALS>

<!-- The actor's advertisement used to publish its capabilities -->
<ADVERTISEMENT>
  <!-- The list of capabilities published by the actor -->
  <CAPABILITIES>
    <CAPABILITY>
      <NAME>CapabilityA</NAME>
      <KEYWORDS>
        <KEYWORD>A</KEYWORD>
        <KEYWORD>B</KEYWORD>
      </KEYWORDS>
      <DESCRIPTION>
        Provides the capability to perform A.
      </DESCRIPTION>
    </CAPABILITY>
  </CAPABILITIES>

  <!-- Other advertisement properties -->
  <PROPERTIES>
  </PROPERTIES>
</ADVERTISEMENT>

<!-- (Optional) The actor's dedicated transport service
     configuration specifying the endpoints through which the
     actor can be contacted, with for each endpoint its
     configuration. If not specified the actor will be
     configured with the transport provided by the hosting
     environment. -->
<TRANSPORT>
  <!-- Other transport properties -->
  <PROPERTIES>
    <!-- Specifies whether this actor wants the actor's transport
         service to queue messages and send these messages in a separate
         thread to the actor (false) or whether this actor wants
         to receive the messages synchronously (true), i.e. indicating
         that the actor queues the messages itself to ensure that
         it does not block the main transport service. -->
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    <PROPERTY name="ci.transport.delivery.synchronous">false</PROPERTY>

    <!-- Specifies the number of times a communication needs to be re-attempted
         when a communication fails. This value serves as the default for
         all communications initiated by this actor. This value can
         be overriden by specifying this property in the envelope of
         a CommunicativeAct for the communication of CommunicativeActs.
         Default is 0 indicating no retries. -->
    <PROPERTY name="ci.communication.qos.recovery.num_retries">3</PROPERTY>

    <!-- Specifies the interval in milliseconds to wait before re-attempting
         a communication after it failed. This value serves as the default for
         all communications initiated by this actor. This value can
         be overriden by specifying this property in the envelope of
         a CommunicativeAct for the communication of CommunicativeActs.
         Default is 1000ms. -->
    <PROPERTY name="ci.communication.qos.recovery.retry_interval">5000</PROPERTY>

  </PROPERTIES>
</TRANSPORT>

<!-- (Optional) The configuration properties controlling data
     distribution service parameters specific for the actor.
-->
<DATA_DISTRIBUTION>
  <PROPERTIES>
    <!-- Specifies the number of times a data publication needs to be re-attempted
         when a publication fails. This value serves as the default for
         all publications initiated by this actor. This value can
         be overriden by specifying this property as part of the QoS
         Properties in a DataInfo record for the publication of Data
         Objects.
         Default is 0 indicating no retries. -->
    <PROPERTY name="ci.communication.qos.recovery.num_retries">3</PROPERTY>

    <!-- Specifies the interval in milliseconds to wait before re-attempting
         a publication after it failed. This value serves as the default for
         all publications initiated by this actor. This value can
         be overriden by specifying this property as part of the QoS
         Properties in a DataInfo record for the publication of Data
         Objects.
         Default is 1000ms. -->
    <PROPERTY name="ci.communication.qos.recovery.retry_interval">5000</PROPERTY>
  </PROPERTIES>
</DATA_DISTRIBUTION>

<!-- General configuration properties -->
<PROPERTIES>
  <!-- For each property the name and value -->

  <!-- The simple filename for the configuration file to load for the actor
       containing the actor's configuration properties. If not specified
       by default the filename will be defined using the actor's simple
       name followed by the cfg extension (ActorA.cfg) -->
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    <PROPERTY name="ci.actor.file.configuration">GUIActor.cfg</PROPERTY>

    <!-- The interval in milliseconds at which to publish the heartbeat/status.
         Leave this property out if no heartbeat is to be published.
    -->
    <PROPERTY name="ci.actor.heartbeat.interval">2500</PROPERTY>

  </PROPERTIES>
</ACTOR>
```

3.1.5 Spring Configuration

Finally to configure the actor in Spring and make it available as a Spring managed bean some configuration elements need to be added to the Spring configuration file used for the Spring application in which the GUI Actor resides. The Spring configuration file for that application is named mas-applicationContext.xml.

First all of the name spaces and schemas to be available in the bean declarations must be declared as part of the beans element. This configures which elements are available in the configuration file. In this configuration file the standard spring framework elements are made available (beans, property, etc), utility elements (for constant references), elements related to defined aspects (AOP), transaction elements, and context elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
```

Since we use the Configurable annotation for our actors we want to make sure that those annotations are processed by Spring when the configuration file is loaded. This is done by adding the element:

```
<context:spring-configured />
```

Next we define the bean for the actual actor.

```
<bean id="guiActorImpl"
    class="gov.nasa.arc.ci.spring.actor.gui.GuiActor"
    abstract="true">
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
  <property name="actorProxy" ref="guiActor" />
  <property name="application" ref="application" />
</bean>
```

As mentioned before this actor must be declared as abstract since Spring does not manage its lifecycle/creation, the CI does. The bean id declared here is the same name passed to the Configurable annotation. In this case 'guiActorImpl'. The class specifies which class provides the actor implementation. The two properties define the dependencies for the actor. This is where Spring's dependency injection comes in. Here we indicate to inject the guiActor bean as the actor's proxy and the application bean for the IUIApplication reference. The reference names specify the bean identifiers assigned to those elements. The declaration for the application will be covered later.

Last we need to declare the proxy that is to be injected into the actor.

```
<bean id="guiActor"
    class="gov.nasa.arc.ci.spring.actor.gui.GuiActorProxy">
</bean>
```

The bean identifier here is the bean identifier used to inject the actor into other Spring managed beans. The class again specifies the implementation class for the proxy. Since this is a Spring managed bean Spring will instantiate this class when this configuration file is processed.

The mas-applicationContext.xml file so far would look as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

  <context:spring-configured />

  <!--          CI ACTOR CONFIGURATION                   -->

  <!--
    - GUI Actor is a CI Actor created by the CI,
    - and is therefore a non-spring bean declared as
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    - abstract. The actor is spring configurable through
    - the use of the Spring Configurable annotation.
    -->
  <bean id="guiActorImpl"
      class="gov.nasa.arc.ci.spring.actor.gui.GuiActor"
      abstract="true">

    <property name="actorProxy" ref="guiActor" />
    <property name="application" ref="application" />
  </bean>

  <!--
    - This is the proxy to the GUI actor that is to be
    - injected into any beans that require a reference
    - to the actor. Other than the beanID nothing must
    - be injected into this bean. Other bean injections
    - must take place in the actual actor implementation.
    -->
  <bean id="guiActor"
      class="gov.nasa.arc.ci.spring.actor.gui.GuiActorProxy">
  </bean>
</beans>
```

This completes the development and configuration of the GUI Actor.

3.2 KFX Parser Actor

The KFX Parser Actor's main responsibility is to provide KFX log parsing services, logging KFX log files and storing and returning those files to whomever is interested.

As with the GUI Actor to develop the KFX Parser Actor and to allow us to make it an injectable bean and to allow the actor to be injected into other beans we need to declare an interface for the actor, develop the implementation, and develop a proxy for the actor.

3.2.1 IKfxParserActor Interface

The IKfxParserActor interface specifies the contract for the actor to the components that wish to use its services. The contract indicates that it depends on the IKfxParser component for controlling the parser and that it provides all standard actor services as defined in the ICIActor interface.

```
package gov.nasa.arc.ci.spring.actor.parser;

import gov.nasa.arc.ci.spring.parser.IKfxParser;
import gov.nasa.ci.common.actor.ICIActor;

public interface IKfxParserActor extends ICIActor {
      public void setParser(IKfxParser parser);
} // IKfxParserActor
```

By using the interface declaration we hide the implementation details from any components that wish to interact with the actor allowing us to easily proxy the actor which is of important to allow us to make the actor injectable into Spring managed beans.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

3.2.2 KfxParserActor Class

The KfxParserActor class provides the implementation of the actor.

The implementation of this actor is very much similar to that of the GUI Actor. In this simple example the parser does not require any initialization or configuration at actor startup. All that needs to be implemented for this actor is the interface method for setting a reference to the parser and to provide an implementation for the required CI actor control and management methods. The configurable annotation is provided the name that matches the bean identifier for the actor in the Spring configuration file.

```
package gov.nasa.arc.ci.spring.actor.parser;

import gov.nasa.arc.ci.spring.parser.IKfxParser;
import gov.nasa.ci.corba.api.ActorException;
import gov.nasa.ci.spring.actor.AbstractSpringCIActor;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("kfxParserActorImpl")
public class KfxParserActor
      extends AbstractSpringCIActor
      implements IKfxParserActor {

  /** The KFX log file parser */
  private IKfxParser m_oParser;

  public void setParser(IKfxParser parser) {
    m_oParser = parser;
  } // setParser

  //
  // ***** Actor Control and Management *****
  //

  public void onInitialize() throws ActorException {
  } // onInitialize

  public void onStart() throws ActorException {
  } // onStart

  public void onStop() throws ActorException {
  } // onStop

  public void onShutdown() {
  } // onShutdown

} // KfxParserActor
```

In the implementation of the actor we take care of implementing support for injecting a reference to an IKfxParser (`setParser`).

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

3.2.3 KfxParserActorProxy Class

In order for the actor to be injectable in other beans we need to also use an actor proxy for this actor. The proxy needs to indicate that it complies with the actor's contract by implementing the IKfxParserActor interface. The CI provides the CIActorProxy base class which provides the implementation for all CI related methods. The actor's proxy then only needs to implement the business methods declared in the actor's interface, in our case `setParser`. All the implementation of this method needs to do is delegate the call to the proxied actor. The CIActorProxy class provides the method `ICIActor getActor()` to get access to the provided actor.

```
package gov.nasa.arc.ci.spring.actor.parser;

import gov.nasa.arc.ci.spring.parser.IKfxParser;
import gov.nasa.ci.common.actor.CIActorProxy;

public class KfxParserActorProxy
       extends CIActorProxy
       implements IKfxParserActor {
  public void setParser(IKfxParser parser) {
    ((IKfxParserActor)getActor()).setParser(parser);
  } // setParser
} // KfxParserActorProxy
```

3.2.4 CI Configuration

The coding of the interface, class, and proxy completes the Java implementation for the actor. To allow the CI Actor Hosting Environment to load and properly configure the actor an actor descriptor must be created. This actor descriptor specifies the name of the Java Class for the actor that needs to be instantiated to create the actor, the simple and qualified names to be assigned to the actor, indicates whether and if so with what names to register the actor in the directory service, optionally defines its capabilities, any service related properties, and actor specific properties.

The following is the actor configuration file for the KFX Parser Actor. The file is named KfxParserActor.ciax. The descriptor file should be placed in the deployment directory configured for the Actor Hosting Environment.

```
<?xml version="1.0" encoding="UTF-8"?>
<ACTOR>
  <!-- The library in which the actor's implementation can be found
       The library element value is the name of the Java
       class that provides the actor implementation.
  -->
  <LIBRARY
type="ci.actor.class">gov.nasa.arc.ci.spring.actor.parser.KfxParserActor</LIBRARY>

  <!-- The credentials of the actor -->
  <CREDENTIALS>
    <!-- The simple name for the actor, used for display or logging
         purposes -->
    <SIMPLENAME>KfxParserActor</SIMPLENAME>
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    <!-- The globally unique fully qualified name of the actor -->
    <QUALIFIEDNAME>gov.nasa.arc.ci.spring.actor.parser.KfxParserActor</QUALIFIEDNAME>

    <!-- The directory name(s) with which the actor is
         registered in the directory service. -->
    <DIRECTORYNAMES>

<DIRECTORYNAME>gov.nasa.arc.ci.spring.actor.parser.KfxParserActor</DIRECTORYNAME>
      <DIRECTORYNAME>ci/spring/actor/KfxParserActor</DIRECTORYNAME>
    </DIRECTORYNAMES>

    <!-- Other credentials properties -->
    <PROPERTIES>
    </PROPERTIES>
  </CREDENTIALS>

  <!-- The actor's advertisement used to publish its capabilities -->
  <ADVERTISEMENT>
    <!-- The list of capabilities published by the actor -->
    <CAPABILITIES>
      <CAPABILITY>
        <NAME>CapabilityA</NAME>
        <KEYWORDS>
          <KEYWORD>A</KEYWORD>
          <KEYWORD>B</KEYWORD>
        </KEYWORDS>
        <DESCRIPTION>
          Provides the capability to perform A.
        </DESCRIPTION>
      </CAPABILITY>
    </CAPABILITIES>

    <!-- Other advertisement properties -->
    <PROPERTIES>
    </PROPERTIES>
  </ADVERTISEMENT>

  <!-- (Optional) The actor's dedicated transport service
       configuration specifying the endpoints through which the
       actor can be contacted, with for each endpoint its
       configuration. If not specified the actor will be
       configured with the transport provided by the hosting
       environment. -->
  <TRANSPORT>
    <!-- Other transport properties -->
    <PROPERTIES>
      <!-- Specifies whether this actor wants the actor's transport
           service to queue messages and send these messages in a separate
           thread to the actor (false) or whether this actor wants
           to receive the messages synchronously (true), i.e. indicating
           that the actor queues the messages itself to ensure that
           it does not block the main transport service. -->
      <PROPERTY name="ci.transport.delivery.synchronous">false</PROPERTY>
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    <!-- Specifies the number of times a communication needs to be re-attempted
         when a communication fails. This value serves as the default for
         all communications initiated by this actor. This value can
         be overriden by specifying this property in the envelope of
         a CommunicativeAct for the communication of CommunicativeActs.
         Default is 0 indicating no retries. -->
    <PROPERTY name="ci.communication.qos.recovery.num_retries">3</PROPERTY>

    <!-- Specifies the interval in milliseconds to wait before re-attempting
         a communication after it failed. This value serves as the default for
         all communications initiated by this actor. This value can
         be overriden by specifying this property in the envelope of
         a CommunicativeAct for the communication of CommunicativeActs.
         Default is 1000ms. -->
    <PROPERTY name="ci.communication.qos.recovery.retry_interval">5000</PROPERTY>

  </PROPERTIES>
</TRANSPORT>

<!-- (Optional) The configuration properties controlling data
     distribution service parameters specific for the actor.
-->
<DATA_DISTRIBUTION>
  <PROPERTIES>
    <!-- Specifies the number of times a data publication needs to be re-attempted
         when a publication fails. This value serves as the default for
         all publications initiated by this actor. This value can
         be overriden by specifying this property as part of the QoS
         Properties in a DataInfo record for the publication of Data
         Objects.
         Default is 0 indicating no retries. -->
    <PROPERTY name="ci.communication.qos.recovery.num_retries">3</PROPERTY>

    <!-- Specifies the interval in milliseconds to wait before re-attempting
         a publication after it failed. This value serves as the default for
         all publications initiated by this actor. This value can
         be overriden by specifying this property as part of the QoS
         Properties in a DataInfo record for the publication of Data
         Objects.
         Default is 1000ms. -->
    <PROPERTY name="ci.communication.qos.recovery.retry_interval">5000</PROPERTY>
  </PROPERTIES>
</DATA_DISTRIBUTION>

<!-- General configuration properties -->
<PROPERTIES>
  <!-- For each property the name and value -->

  <!-- The simple filename for the configuration file to load for the actor
       containing the actor's configuration properties. If not specified
       by defailt the filename will be defined using the actor's simple
       name followed by the cfg extension (ActorA.cfg) -->
  <PROPERTY name="ci.actor.file.configuration">KFXActor.cfg</PROPERTY>
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    <!-- The interval in milliseconds at which to publish the heartbeat/status.
         Leave this property out if no heartbeat is to be published.
    -->
    <PROPERTY name="ci.actor.heartbeat.interval">2500</PROPERTY>

  </PROPERTIES>
</ACTOR>
```

3.2.5 Spring Configuration

Finally to configure the actor in Spring and make it available as a Spring managed bean some configuration elements need to be added to the Spring configuration file used for the Spring application in which the KFX Parser Actor resides. The Spring configuration file for that application is named oca-applicationContext.xml.

First all of the name spaces and schemas to be available in the bean declarations must be declared as part of the beans element. This configures which elements are available in the configuration file. In this configuration file the standard spring framework elements are made available (beans, property, etc), utility elements (for constant references), elements related to defined aspects (AOP), transaction elements, and context elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
```

Since we use the Configurable annotation for our actors we want to make sure that those annotations are processed by Spring when the configuration file is loaded. This is done by adding the element:

```
<context:spring-configured />
```

Next we define the bean for the actual actor.

```
<bean id="kfxParserActorImpl"
    class="gov.nasa.arc.ci.spring.actor.parser.KfxParserActor"
    abstract="true">

  <property name="actorProxy" ref="kfxParserActor" />
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
  <property name="messageHandlerFactory"
       ref="kfxParserActorMessageHandlerFactory" />
  <property name="parser" ref="kfxParser" />
</bean>
```

As mentioned before this actor must be declared as abstract since Spring does not manage its lifecycle/creation, the CI does. The bean id declared here is the same name passed to the Configurable annotation. In this case 'kfxParserActorImpl'. The class specifies which class provides the actor implementation. The three properties define the dependencies for the actor. This is where Spring's dependency injection comes in. Here we indicate to inject the kfxParserActor bean as the actor's proxy, a message handler factory discussed later, and the parser bean for the IKfxParser reference. The reference names specify the bean identifiers assigned to those elements. The declaration for the message handler factory and parser will be covered later.

Last we need to declare the proxy that is to be injected into the actor.

```
<bean id="kfxParserActor"
    class="gov.nasa.arc.ci.spring.actor.parser.KfxParserActorProxy">
</bean>
```

The bean identifier here is the bean identifier used to inject the actor into other Spring managed beans. The class again specifies the implementation class for the proxy. Since this is a Spring managed bean Spring will instantiate this class when this configuration file is processed.

The oca-applicationContext.xml file so far would look as follows.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

  <context:spring-configured />

  <!--          CI ACTOR CONFIGURATION                     -->

  <!--
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    - KFX Parser Actor is a CI Actor created by the CI,
    - and is therefore a non-spring bean declared as
    - abstract. The actor is spring configurable through
    - the use of the Spring Configurable annotation.
    -->
  <bean id="kfxParserActorImpl"
    class="gov.nasa.arc.ci.spring.actor.parser.KfxParserActor"
    abstract="true">

    <property name="actorProxy" ref="kfxParserActor" />
    <property name="messageHandlerFactory"
ref="kfxParserActorMessageHandlerFactory" />
    <property name="parser" ref="kfxParser" />
  </bean>

  <!--
    - This is the proxy to the KFX Parser actor that is to be
    - injected into any beans that require a reference
    - to the actor. Other than the beanID nothing must
    - be injected into this bean. Other bean injections
    - must take place in the actual actor implementation.
    -->
  <bean id="kfxParserActor"
      class="gov.nasa.arc.ci.spring.actor.parser.KfxParserActorProxy">
  </bean>
</beans>
```

This completes the development and configuration of the KFX Parser Actor.

4 Data Services

In this simple application the only data to be stored and accessed are the parsed KFX File records. The KFX File records describe data about the files that were transferred between ground operations and the International Space Station.

For the data storage we decided to use a relational database and chose MySQL as the DBMS. To allow our application to work with any DBMS we decided to add an object relational mapping (ORM) solution that takes care of managing how the Java objects with the data get stored in the DBMS. In our example we have chosen the popular Hibernate tool. Spring provides full support for Hibernate and allows us to fully configure Hibernate with the MySQL DBMS as the data store.

To make our application flexible with regards to the persistence mechanism we are using the Data Access Object design pattern. The Data Access Objects implement the specifics about how to interact with the data store or ORM tool. The business services interface with the DAO objects to interact with the data store. If there is a need to change the data store we only need to swap out the DAO objects for the DAO objects that interact with the new data store and the code business logic remains unchanged.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

Last we developed a KFX File Service that provides all the KFX File related services needed by the application such as adding files to the data store and retrieving files from the data store. This file service uses the DAO for KFX Files to access the actual data store.

The data services are shared across the distributed system. The services themselves are not distributed, but the configuration is the same for both. The various components that provide all of the data services are all created, managed, and configured using Spring.

4.1 KFX File Data Representation

For the representation of a KFX File record we defined a KFXFile Class. This class is a simple Serializable POJO. There is nothing special about the data except for the key. The key will be used as the unique identifier for storing the data in the data store. This key is made up of a combination of the fields in the class, namely the action date, action, filename, client, and line number. The class is declared as follows:

```
package gov.nasa.arc.ci.spring.common;

import java.io.Serializable;
import java.util.Date;

public class KfxFile implements Serializable {

  /**
   * Empty constructor, required for persistence layer.
   */
  public KfxFile() {
  } // KfxFile

  /**
   * Constructor, creates a new KfxFile.
   */
  public KfxFile(
      String client,
      int lineNumber,
      Action fileAction,
      Date actionTime,
      String filename,
      String fileExtension,
      int fileSize,
      String clientDirectory) {
    m_sClient = client;
    m_nLineNumber = lineNumber;
    m_oFileAction = fileAction;
    m_oActionTime = actionTime;
    m_sFileName = filename;
    m_sFileExtension = fileExtension;
    m_nFileSize = fileSize;
    m_sClientDirectory = clientDirectory;
    m_sKey = createKey();
  } // KfxFile

  private String              m_sClient;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
private int                m_nLineNumber;
private Action             m_oFileAction;
private Date               m_oActionTime;
private String             m_sFileName;
private String             m_sFileExtension = "";
private int                m_nFileSize = 0;
private String             m_sClientDirectory;
private String             m_sKey;

public void setClient(String client) {
  m_sClient = client;
} // setClient

public String getClient() {
  return m_sClient;
} // getClient

public void setLineNumber(int lineNumber) {
  m_nLineNumber = lineNumber;
} // setLineNumber

public int getLineNumber() {
  return m_nLineNumber;
} // getLineNumber

public void setFileAction(Action action) {
  m_oFileAction = action;
} // setFileAction

public Action getFileAction() {
  return m_oFileAction;
} // getFileAction

public void setActionTime(Date time) {
  m_oActionTime = time;
} // getActionTime

public Date getActionTime() {
  return m_oActionTime;
} // getActionTime

/**
 * Sets the file name including extension excluding path.
 *
 * @param fileName  The file name
 */
public void setFileName(String fileName) {
  m_sFileName = fileName;
  m_sKey = null;
} // setFileName

public String getFileName() {
  return m_sFileName;
} // getFileName
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
  public void setFileExtension(String fileExtension) {
    m_sFileExtension = fileExtension;
  } // setFileExtension

  public String getFileExtension() {
    return m_sFileExtension;
  } // getFileExtension

  public void setFileSize(int size) {
    m_nFileSize = size;
  } // setFileSize;

  public int getFileSize() {
    return m_nFileSize;
  } // getFileSize;

  public void setClientDirectory(String dir) {
    if (dir != null && !dir.equals("")) {
      if (!(dir.endsWith("/") || dir.endsWith("\\"))) {
        dir += "/";
      } // end if
      dir = dir.replace('\\', '/');
    } // end if
    m_sClientDirectory = dir;
  } // setClientDirectory

  public String getClientDirectory() {
    return m_sClientDirectory;
  } // getClientDirectory

  public void setKey(String key) {
    m_sKey = key;
  } // setKey

  public String getKey() {
    if (m_sKey == null) {
      m_sKey = createKey();
    } // end if
    return m_sKey;
  } // getKey

  /**
   * Returns a formatted unique identifier string for this file -
   * "<ActionDate>_<ACTION>_<FileName>_<ClientID>_<LineNumber>".
   *
   * @return String a unique identifier.
   */
  public String createKey() {
    StringBuffer buffer = new
StringBuffer(Long.toString(getActionTime().getTime()));
    buffer.append('_');
    buffer.append(getFileAction());
    buffer.append('_');
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    buffer.append(getFileName());
    buffer.append('_');
    buffer.append(getClient());
    buffer.append('_');
    buffer.append(getLineNumber());
    return buffer.toString();
  } // createKey

  @Override
  public int hashCode() {
    return getKey().hashCode();
  } // hashCode

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof KfxFile) {
      return hashCode() == obj.hashCode();
    } else {
      return false;
    } // end if
  } // equals

} // KfxFile
```

Note that it is important to define the hashCode and equals methods for the data classes in general, but specifically in this case since the ORM tool we chose requires these methods to be properly defined. Hibernate recommends against using the unique identifier of an object as the basis for the hashCode and equals methods in the case an auto-generated identifier is used. When using a self-created key this restriction does not apply.

The KfxFile uses the Action enum type which is declared as:

```
package gov.nasa.arc.ci.spring.common;

public enum Action {
  Uplink,
  Downlink,
  Delete,
  Copy,
  Move;
} // Action
```

One other important note to make is that the data class just represents the 'business' data and does not contain any methods or annotations to support the storage of this data. This data representation is completely independent of how the data will be stored.

Since this class is only used for internal data representation there is no configuration required for the CI or for Spring.

4.2 Data Store

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

In our example we use MySQL as our data store. In our example we use a MySQL database named SpringCIDB. We use MySQL 5.x in this example. To create the database follow these instructions (Windows):

1. Start a command prompt
2. Run:
   >mysql -u root -p
   Enter your root password. The prompt should change to:
   mysql>
3. Enter the following mysql commands to create the database, user (uid: brahms, pwd: brahms) and give the user the appropriate privileges:
   mysql> create database SpringCIDB;
   mysql> GRANT ALL PRIVILEGES ON SpringCIDB.* TO 'brahms'@'localhost' IDENTIFIED BY 'brahms' WITH GRANT OPTION;

The database has now been created and is ready for use.

To allow hibernate to access this data store we need to declare a data source that defines how the data store can be accessed. No Java coding is required for this task. The data source is fully configured in Spring. To enhance performance in applications accessing data stores it is highly recommended to use database connection pools. In our application we use the Apache Commons database connection pool.

The configuration of the data source is declared as following in our mas-applicationContext.xml and oca-applicationContext.xml files:

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${database.driver}" />
  <property name="url" value="${database.url}" />
  <property name="username" value="${database.username}" />
  <property name="password" value="${database.password}" />
  <property name="initialSize" value="${database.initialConnections}" />
  <property name="maxActive" value="${database.maxConnections}" />
</bean>
```

The data source is defined as a Spring managed bean with the BasicDataSource as the data source class from the Apache Commons package which provides a connection pool to the data store. The listed properties are injected by Spring. In this configuration all the values for these properties are stored in an external configuration file named jdbc.properties which must be located on the classpath of the application. This properties file's contents is:

```
database.url=jdbc:mysql://localhost/SpringCIDB
database.driver=com.mysql.jdbc.Driver
database.username=brahms
database.password=brahms
database.initialConnections=5
database.maxConnections=10
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

The URL property specifies the name of the database and the host on which it resides. The driver is the MySQL JDBC driver used to access the database from Java. We use the MySQL Connector for Java (version 5.1.6). The rest are configuration for authenticating against the database and properties to configure the connection pool.

We now need to make those properties with their values available to Spring or otherwise Spring would not be able to substitute the variables in the data source configuration. For this we need to create a property configurer bean that loads the desired properties files.

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      <list>
        <value>jdbc.properties</value>
      </list>
    </property>
</bean>
```

The Spring Framework provides a class that takes care of loading these types of configuration files, the PropertyPlaceholderConfigurer is this class. In this bean definition the jdbc.properties file is listed as a file to be loaded. The file needs to be located in the classpath.

4.3 Object Relational Mapping with Hibernate

Hibernate applications make use of mapping files containing meta data defining object/relational mappings for Java classes. A mapping file is designated with a suffix of .hbm.xml. Within each configuration file, classes to be made persistent are mapped to database tables and properties are defined which map class-fields to columns and primary keys. The following is the content of the Hibernate configuration file for the KfxFile class declared in the file KfxFile.hbm.xml located in the package gov/nasa/arc/ci.spring/db/hibernate.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="gov.nasa.arc.ci.spring.db.hibernate">

  <typedef name="Action"
class="gov.nasa.arc.ci.spring.db.hibernate.EnumUserType">
    <param name="enumClass">gov.nasa.arc.ci.spring.common.Action</param>
  </typedef>

  <class name="gov.nasa.arc.ci.spring.common.KfxFile" table="KFXFILES">
    <id name="key" column="KFXFILE_ID">
      <generator class="assigned" />
    </id>
    <property name="client" column="CLIENT" />
    <property name="lineNumber" column="LINE_NUMBER" />
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    <property name="fileAction" column="ACTION" type="Action" />
    <property name="actionTime" column="ACTION_TIME" type="timestamp" />
    <property name="fileName" column="FILE_NAME" />
    <property name="fileExtension" column="FILE_EXTENSION" />
    <property name="fileSize" column="FILE_SIZE" />
    <property name="clientDirectory" column="CLIENT_DIRECTORY" />
  </class>

</hibernate-mapping>
```

As part of the hibernate mapping we specify the package in which the hbm.xml file is located. The class element defines the mapping for our KfxFile. The name mist match the qualified name of our data class. The table attribute defines the name of the table that stores the files, KFXFILES in this case. The id field specifies which field is used as the primary key. The column attribute defined the name of the column in the KFXFILES table. As part of the id element you can specify the type of generator to use. Hibernate provides a variety of possible id generators also supporting database generated id's. In our case we generate the id/key ourselves and use the 'assigned' generator. This is the default if not specified. The remaining properties define the mappings for the remaining fields in the data class. Hibernate chooses the most appropriate type mappings for the fields in a class based on the declared type for that field. To ensure the proper type is used you can specify the type as part of the property declaration. In our case we indicate that the actionTime property must be of type timestamp. A special case is the fileAction mapping. We use the Action enum type for that field. Hibernate does not offer a mapping for Java enumerations so you need to provide one yourself. To provide this mapping yourself you need to write a custom Hibernate UserType. Fortunately there are plenty of custom user type implementations available on the web that support Java enum types. We copied one from Martin Kersten. To make this mapping available to Hibernate we just need to declare a typedef for the Action type and specify the class for the user type that handles the mapping. The only argument required for our mapping is the enumClass which specifies the qualified name for the Action enum class. By default this enumeration is mapped to a String which is mapped to a VARCHAR by Hibernate.

The class definition for the EnumUserType is as follows (for a detailed description and other versions of this class see https://www.hibernate.org/272.html):

```
package gov.nasa.arc.ci.spring.db.hibernate;

import java.io.Serializable;
import java.lang.reflect.Method;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;

import org.hibernate.HibernateException;
import org.hibernate.type.NullableType;
import org.hibernate.type.TypeFactory;
import org.hibernate.usertype.ParameterizedType;
import org.hibernate.usertype.UserType;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
/**
 * Implements a generic enum user type identified/represented
 * by a single identifier/column.
 * <p>
 * <ul>
 *    <li>The enum type being represented by the certain user type must be set
 *        by using the 'enumClass' property.</li>
 *    <li>The identifier representing a enum value is retrieved by the
 *        identifierMethod. The name of the identifier method can be specified
 *        by the 'identifierMethod' property and by default the name() method
 *        is used.</li>
 *    <li>The identifier type is automatically determined by
 *        the return-type of the identifierMethod.</li>
 *    <li>The valueOfMethod is the name of the static factory method returning
 *        the enumeration object being represented by the given indentifier.
 *        The valueOfMethod's name can be specified by setting the
 *        'valueOfMethod' property. The default valueOfMethod's name is
 *        'valueOf'.</li>
 * </ul>
 * </p>
 * <p>
 * Example of an enum type represented by an int value:
 * <code>
 * public enum SimpleNumber {
 *   Unknown(-1), Zero(0), One(1), Two(2), Three(3);
 *   int value;
 *   protected SimpleNumber(int value) {
 *     this.value = value;
 *   }
 *
 *   public int toInt() { return value; }
 *   public SimpleNumber fromInt(int value) {
 *     switch(value) {
 *       case 0: return Zero;
 *       case 1: return One;
 *       case 2: return Two;
 *       case 3: return Three;
 *       default:
 *         return Unknown;
 *     }
 *   }
 * }
 * </code>
 * <p>
 * The Mapping would look like this:
 * <code>
 *    <typedef name="SimpleNumber"
 *           class="gov.nasa.ci.spring.db.hibernate.EnumUserType">
 *        <param name="enumClass">SimpleNumber</param>
 *        <param name="identifierMethod">toInt</param>
 *        <param name="valueOfMethod">fromInt</param>
 *    </typedef>
 *    <class ...>
 *      ...
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
 *      <property name="number" column="number" type="SimpleNumber"/>
 *    </class>
 * </code>
 *
 * @author Martin Kersten, Ron van Hoof
 * @version $Revision: 1.1.1.1 $ $Date: 2009/12/15 22:47:04 $ $Author: rvhoof $
 */
public class EnumUserType implements UserType, ParameterizedType {

  /*
   * Constants
   */
  /** Default name of the method used to obtain the unique identifier for
       the enum */
  private static final String DEFAULT_IDENTIFIER_METHOD_NAME = "name";
  /** Default name of the method used to obtain the enum value */
  private static final String DEFAULT_VALUE_OF_METHOD_NAME = "valueOf";

  /*
   * Member Variables
   */
  /** The Enum class */
  private Class<? extends Enum> m_oEnumClass;
  /** The type of the identifier for the enum */
  private Class<?> m_oIdentifierType;
  /** The method used to obtain the identifier for the enum */
  private Method m_oIdentifierMethod;
  /** The method used to obtain the Enum value */
  private Method m_oValueOfMethod;
  /** The NullableType for the identifier type */
  private NullableType m_oType;
  /** The SQL type mapping for the identifier type */
  private int[] m_anSqlTypes;

  /*
   * Member Operations
   */
  /**
   * Set the parameter values to configure this EnumUserType.
   *
   * @param parameters the parameters
   */
  public void setParameterValues(Properties parameters) {
    String enumClassName = parameters.getProperty("enumClass");
    try {
      m_oEnumClass = Class.forName(enumClassName).asSubclass(Enum.class);
    } catch (ClassNotFoundException cfne) {
      throw new HibernateException("Enum class not found", cfne);
    } // end try

    String identifierMethodName = parameters.getProperty(
       "identifierMethod",
       DEFAULT_IDENTIFIER_METHOD_NAME);
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    try {
      m_oIdentifierMethod = m_oEnumClass.getMethod(
              identifierMethodName, new Class[0]);
      m_oIdentifierType = m_oIdentifierMethod.getReturnType();
    } catch (Exception e) {
      throw new HibernateException("Failed to obtain identifier method", e);
    } // end try

    m_oType = (NullableType) TypeFactory.basic(m_oIdentifierType.getName());

    if (m_oType == null) {
      throw new HibernateException("Unsupported identifier type " +
              m_oIdentifierType.getName());
    } // end if

    m_anSqlTypes = new int[] { m_oType.sqlType() };

    String valueOfMethodName = parameters.getProperty(
       "valueOfMethod",
       DEFAULT_VALUE_OF_METHOD_NAME);

    try {
      m_oValueOfMethod = m_oEnumClass.getMethod(
              valueOfMethodName, new Class[] { m_oIdentifierType });
    } catch (Exception e) {
      throw new HibernateException("Failed to obtain valueOf method", e);
    } // end try
  } // setParameterValues

  /**
   * Return the SQL type codes for the columns mapped by this type. The
   * codes are defined on <tt>java.sql.Types</tt>.
   * @see java.sql.Types
   * @return int[] the typecodes
   */
  public int[] sqlTypes() {
    return m_anSqlTypes;
  } // sqlTypes

  /**
   * The class returned by <tt>nullSafeGet()</tt>.
   *
   * @return Class
   */
  public Class<?> returnedClass() {
    return m_oEnumClass;
  } // returnedClass

  /**
   * Compare two instances of the class mapped by this type for persistence
   * "equality". Equality of the persistent state.
   *
   * @param x the first instance
   * @param y the second instance
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
 * @return boolean true if equals, false otherwise
 * @throws HibernateException
 */
public boolean equals(Object x, Object y) throws HibernateException {
  return x == y;
} // equals

/**
 * Get a hashcode for the instance, consistent with persistence "equality"
 *
 * @param x the Object for which to obtain the hash code
 * @return int the hashcode
 * @throws HibernateException
 */
public int hashCode(Object x) throws HibernateException {
  return x.hashCode();
} // hashCode

/**
 * Retrieve an instance of the mapped class from a JDBC resultset.
 * Implementors should handle possibility of null values.
 *
 * @param rs a JDBC result set
 * @param names the column names
 * @param owner the containing entity
 * @return Object
 * @throws HibernateException
 * @throws SQLException
 */
public Object nullSafeGet(ResultSet rs, String[] names, Object owner)
     throws HibernateException, SQLException {
  Object identifier = m_oType.get(rs, names[0]);
  if (rs.wasNull()) {
    return null;
  } // end if

  try {
    return m_oValueOfMethod.invoke(
           m_oEnumClass,
           new Object[] { identifier });
  } catch (Exception e) {
    throw new HibernateException("Exception while invoking valueOf method '"
           + m_oValueOfMethod.getName() + "' of " +
           "enumeration class '" + m_oEnumClass + "'", e);
  } // end try
} // nullSafeGet

/**
 * Write an instance of the mapped class to a prepared statement.
 * Implementors should handle possibility of null values. A multi-column
 * type should be written to parameters starting from <tt>index</tt>.
 *
 * @param st a JDBC prepared statement
 * @param value the object to write
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
 * @param index statement parameter index
 * @throws HibernateException
 * @throws SQLException
 */
public void nullSafeSet(PreparedStatement st, Object value, int index)
     throws HibernateException, SQLException {
  try {
    if (value == null) {
      st.setNull(index, m_oType.sqlType());
    } else {
      Object identifier = m_oIdentifierMethod.invoke(value, new Object[0]);
      m_oType.set(st, identifier, index);
    } // end if
  } catch (Exception e) {
    throw new HibernateException(
           "Exception while invoking identifierMethod '" +
           m_oIdentifierMethod.getName() + "' of " +
           "enumeration class '" + m_oEnumClass + "'", e);
  } // end try
} // nullSafeSet

/**
 * Return a deep copy of the persistent state, stopping at entities and at
 * collections. It is not necessary to copy immutable objects, or null
 * values, in which case it is safe to simply return the argument.
 *
 * @param value the object to be cloned, which may be null
 * @return Object a copy
 */
public Object deepCopy(Object value) throws HibernateException {
  return value;
} // deepCopy

/**
 * Are objects of this type mutable?
 *
 * @return boolean true if mutable, false otherwise
 */
public boolean isMutable() {
  return false;
} // isMutable

/**
 * Transform the object into its cacheable representation. At the very least
 * this method should perform a deep copy if the type is mutable. That may
 * not be enough for some implementations, however; for example, associations
 * must be cached as identifier values. (optional operation)
 *
 * @param value the object to be cached
 * @return a cachable representation of the object
 * @throws HibernateException
 */
public Serializable disassemble(Object value) throws HibernateException {
  return (Serializable) value;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    } // disassemble

    /**
     * Reconstruct an object from the cacheable representation. At the very least
     * this method should perform a deep copy if the type is mutable.
     * (optional operation)
     *
     * @param cached the object to be cached
     * @param owner the owner of the cached object
     * @return a reconstructed object from the cachable representation
     * @throws HibernateException
     */
    public Object assemble(Serializable cached, Object owner)
         throws HibernateException {
      return cached;
    } // assemble

    /**
     * During merge, replace the existing (target) value in the entity we
     * are merging to with a new (original) value from the detached entity we
     * are merging. For immutable objects, or null values, it is safe to
     * simply return the first parameter. For mutable objects, it is safe to
     * return a copy of the first parameter. For objects with component values,
     * it might make sense to recursively replace component values.
     *
     * @param original the value from the detached entity being merged
     * @param target the value in the managed entity
     * @return the value to be merged
     */
    public Object replace(Object original, Object target, Object owner)
         throws HibernateException {
      return original;
    } // replace

  } // EnumUserType
```

4.4 KFX File Data Access Object

To make our application flexible with regards to the persistence mechanism we are using the Data Access Object design pattern. The Data Access Objects implement the specifics about how to interact with the data store or ORM tool. The business services interface with the DAO objects to interact with the data store. If there is a need to change the data store we only need to swap out the DAO objects for the DAO objects that interact with the new data store and the code business logic remains unchanged.

4.4.1 KFX File DAO Interface

As for actors we will use a contract based design for the DAO classes to allow us to easily develop different implementations of the DAO classes that support different data storage solutions. For our application we declared the IKfxFileDao interface.

```
package gov.nasa.arc.ci.spring.db;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
import org.springframework.dao.DataAccessException;
import org.springframework.orm.ObjectRetrievalFailureException;

import gov.nasa.arc.ci.spring.common.KfxFile;

public interface IKfxFileDao {

  /**
   * Looks up the KfxFile with the specified unique identifier/key.
   *
   * @param id the unique key for the requested KfxFile
   * @return KfxFile the file if found
   * @throws ObjectRetrievalFailureException if not found
   * @throws DataAccessException in case of Hibernate errors
   */
  public KfxFile find(String id);

  /**
   * Saves the specified KfxFile into the database creating
   * a record for the file if it does not yet exist and
   * updating it if it does exist.
   *
   * @param file the KfxFile to save
   * @throws DataAccessException in case of Hibernate errors
   */
  public void save(KfxFile file);

  /**
   * Removes the specified KfxFile from the database.
   *
   * @param file the KfxFile to remove
   * @throws DataAccessException in case of Hibernate errors
   */
  public void remove(KfxFile file);

  /**
   * Removes the KfxFile with the specified unique identifier/key
   * from the database.
   *
   * @param id the unique identifier for the KfxFile to remove
   * @throws DataAccessException in case of Hibernate errors
   */
  public void removeById(String id);

} // IKfxFileDao
```

Note that the Spring framework prides itself on wrapping exceptions into its own exceptions. The Spring exceptions are all runtime exceptions. The designers of the Spring framework are of the opinion that for most exceptions no decent recovery is possible and they should therefore be defined as runtime exceptions that are handled higher up in the application logic. This is the same for all database related exceptions. The application developer can choose to handle an exception but is not required to do so keeping the application code simpler in most cases.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

Note that Spring will automatically rollback any database transactions during which time a runtime exception is raised.

4.4.2 Hibernate Session Factory

Before getting into the implementation of the DAO class for the KFX File for Hibernate we first need to cover some basic aspects of Hibernate's persistence mechanism. The main interface for interacting with Hibernate is org.hibernate.Session. The Session object provides basic data access functionality such as the ability to save, update, delete, and load objects from a database. It is through the Hibernate Session that an application's DAO will perform all of its persistence needs.

The standard way to get a reference to a Hibernate Session object is through an implementation of Hibernate's SessionFactory interface. Among other things, SessionFactory is responsible for opening, closing, and managing Hibernate Sessions.

A SessionFactory is required to configure the DAO classes. The session factory can be configured in its entirety in the Spring configuration file. The following bean configuration needs to be added to our applicationContext.xml files.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>gov/nasa/arc/ci/spring/db/hibernate/KfxFile.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
      <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
    </props>
  </property>
</bean>
```

In our case we use the LocalSessionFactoryBean for our session factory. If we would have used annotations in our DAO's we would have needed to use the AnnotationSessionFactoryBean.

The session factory needs to be injected with the data source we defined earlier. All we need to do is refer to the bean id of that data source. We also need to indicate all of the mapping files that need to be interpreted by Hibernate. Here we list the path to the KfxFile.hbm.xml file as it can be found on the classpath (in the jar file of the application). Last we specify some hibernate configuration settings. One being the dialect which is configured in an external configuration file named hibernate.properties. The hbm2dll.auto property is used to specify whether hibernate should automatically manipulate the database tables. In this case we set it up as create-drop which has Hibernate drop all tables and recreate them fresh everytime the application is started. This option should be listed in only one of the applicationContext.xml files (in our case in the file for MAS). A value of update would cause Hibernate to verify whether any changes were made to the data definition language for the database tables and if so

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

update the tables to match the updated DDL and do nothing if the DDL remained the same. In production you would not use this property, since the database creation or updates would be performed using scripts to ensure that no data is lost. The possible values for this property are:

- *validate*: validate the database schema, makes no changes to the database.
- *update*: update the schema
- *create*: creates the schema (tables, referential integrity, etc)
- *create-drop*: drops the schema at the end of the session

The hibernate.properties file has as contents:

```
hibernate.dialect=org.hibernate.dialect.MySQLInnoDBDialect
```

Indicating that we are using a MySQL database and want to use InnoDB tables (MyISAM does not support transaction isolation levels). As with the jdbc.properties file we will need to configure Spring to load this properties file. All we need to do is add a the hibernate.properties file as a value to the file list for the property configurer.

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      <list>
        <value>jdbc.properties</value>
        <value>hibernate.properties</value>
      </list>
    </property>
</bean>
```

For a complete list of all of the possible Hibernate configuration properties see the Hibernate reference documentation [4] section 3. This takes care of the creation and configuration of the Hibernate Session Factory.

4.4.3 Hibernate KFX File DAO Class

As mentioned before when using Hibernate you have to obtain a Session from a `SessionFactory` and use it to interface with Hibernate. Spring provides a `HibernateTemplate` that provides an abstract layer over a Hibernate Session. HibernateTemplate's main responsibility is to simplify the work of opening and closing Hibernate Sessions and to convert Hibernate specific exceptions to one of the Spring ORM exceptions.

To simplify the configuration of DAO classes Spring offers the `HibernateDaoSupport` class. This `HibernateDaoSupport` only requires the injection of a session factory and manages the creation of an `HibernateTemplate` so that there is no need to explicitly configure a `HibernateTemplate` in the Spring configuration file and having to inject it in the DAO classes. DAO classes use the `HibernateTemplate` managed by this `HibernateDaoSupport` to interface with Hibernate. To invoke the appropriate Hibernate methods for saving, updating, retrieving, and deleting objects obtain a reference to the hibernate template using the `getHibernateTemplate()` method made available

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

by `HibernateDaoSupport` and invoking the desired method. Note that the `saveOrUpdate` method checks whether the object was already stored or not, if not it will be stored as a new record, if it was already stored it will be updated. There are two types of methods to obtain an object from the data store, get and load. The get method returns null if the object is not found, whereas the load method raises an exception if the object is not found. In this case we chose to use load to allow us to deal with missing objects via exception handling. Our Dao class could be modified to support two types of find methods as well, a get and a find method with the get returning null and find raising the exception to give the application developer the option to choose either depending on the needs of the application. In our example there was no need for the separation.

The implementation of the DAO class for the KFX File is now as simple as:

```
package gov.nasa.arc.ci.spring.db.hibernate;

import gov.nasa.arc.ci.spring.common.KfxFile;
import gov.nasa.arc.ci.spring.db.IKfxFileDao;

import org.springframework.dao.DataAccessException;
import org.springframework.orm.ObjectRetrievalFailureException;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

public class KfxFileDao extends HibernateDaoSupport implements IKfxFileDao {

  public KfxFile find(String id) {
    return (KfxFile)getHibernateTemplate().load(KfxFile.class, id);
  } // find

  public void save(KfxFile file) {
    getHibernateTemplate().saveOrUpdate(file);
  } // save

  public void remove(KfxFile file) {
    getHibernateTemplate().delete(file);
  } // remove

  public void removeById(String id) {
    KfxFile oFile = find(id);
    remove(oFile);
  } // removeById

} // KfxFileDao
```

4.4.4 DAO Spring Configuration

The last thing to do is to make the DAO class for KFX Files available to those beans that need to manage the storage of KFX Files. We need to add a configuration for the KFX File DAO bean and inject it with the session factory configured before. Add this configuration to both applicationContext.xml files.

```
<bean id="kfxFileDao"
    class="gov.nasa.arc.ci.spring.db.hibernate.KfxFileDao">
  <property name="sessionFactory" ref="sessionFactory" />
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
</bean>
```

4.5 KFX File Service

One thing to note about DAO classes is that they do not deal with any transactions. The methods do not automatically commit or rollback. This is by design since it is easily possible that as part of one atomic transaction you would need to retrieve an object, associate it with another object, then save the object requiring two or more interactions with DAO objects. If each interaction with a DAO object would be automatically committed the overall transaction would not be atomic.

Transactions are covered in more detail in the next section. To allow us to properly manage transactions we develop a service with as responsibility to manage all KFX File related operations. These operations could involve multiple interactions with a single DAO and/or involve multiple interactions with multiple DAO's. To allow us to declaratively define our transactions each atomic operation is defined as a method in the service.

4.5.1 IKfxFileService Interface

Since we are designing by contract as recommended by Spring and since it is generally a good design practice we define the IKfxFileService interface as:

```
package gov.nasa.arc.ci.spring.common;

import java.util.Set;

public interface IKfxFileService {

  /**
   * Adds the specified KfxFile to storage.
   *
   * @param file the KfxFile to add
   * @throws org.springframework.dao.DataAccessException if the file could
   *         not be persisted
   */
  public void addKfxFile(KfxFile file);

  /**
   * Retrieves the KfxFile with the specified key.
   *
   * @param key the unique identifier for the requested KfxFile
   * @return KfxFile the requested KfxFile, null if not found
   * @throws org.springframework.dao.DataAccessException if there was a
   *         database problem
   */
  public KfxFile getKfxFile(String key);

  /**
   * Retrieves the KFXFiles with the specified keys.
   *
   * @param keys the unique identifiers for the requested KFXFiles
   * @return Set<KfxFile> the KFXFiles that were found
   * @throws org.springframework.orm.ObjectRetrievalFailureException if a
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
     *         file for a key could not be found
     * @throws org.springframework.dao.DataAccessException if there was a
     *         database problem
     */
    public Set<KfxFile> getKfxFiles(Set<String> keys);

  } // IKfxFileService
```

Also in this service we show that all exceptions raised are runtime exceptions. If it makes sense for the application design it would be possible to raise a checked business exception and wrap the actual cause for the problem into that exception.

4.5.2 KfxFileService Class

Our KFX File Service will be using the KFX File DAO to interact with Hibernate and to implement the services. Again we will use Spring to have it inject the appropriate DAO into this class. So support this we need to add a setter method for the DAO object. The implementation of the service methods use that DAO to store and obtain the data.

```
package gov.nasa.arc.ci.spring.common;

import gov.nasa.arc.ci.spring.db.IKfxFileDao;

import java.util.HashSet;
import java.util.Set;

import org.springframework.orm.ObjectRetrievalFailureException;

public class KfxFileService implements IKfxFileService {
  /** The DAO used to persist and retrieve KFXFiles. */
  private IKfxFileDao m_oKfxFileDao;

  /**
   * Sets the KfxFile DAO used to persist and retrieve KFXFiles.
   *
   * @param dao the IKfxFileDao
   */
  public void setKfxFileDao(IKfxFileDao dao) {
    m_oKfxFileDao = dao;
  } // setKfxFileDao

  public void addKfxFile(KfxFile file) {
    m_oKfxFileDao.save(file);
  } // addKfxFile

  public KfxFile getKfxFile(String key) {
    try {
      return m_oKfxFileDao.find(key);
    } catch (ObjectRetrievalFailureException orfx) {
      return null;
    } // end try
  } // getKfxFile
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    public Set<KfxFile> getKfxFiles(Set<String> keys) {
      Set<KfxFile> coFiles = new HashSet<KfxFile>();
      for (String sKey : keys) {
        coFiles.add(m_oKfxFileDao.find(sKey));
      } // end for
      return coFiles;
    } // getKfxFiles
  } // KfxFileService
```

In this implementation we show that `addKfxFile` is used only to add new objects by using the save method and not `saveOrUpdate`. We would add an `updateKfxFile` method to the service if we also want to support updating of KfxFiles. In our example there is no need for it. The `getKfxFile` method shows us handling the runtime `ObjectRetrievalFailureException` when an object is not found since we want the service to return null if not found. The `getKfxFiles` method is an example of us calling a method on a DAO object multiple times. The method raises an `ObjectRetrievalFailureException` if an object for one of the keys could not be found.

4.5.3 Spring Configuration

The last thing for us to do is to make the service available as a Spring managed bean and to configure it. For this we add the following configuration to each of the applicationContext.xml files.

```
<bean id="kfxFileService"
    class="gov.nasa.arc.ci.spring.common.KfxFileService">
  <property name="kfxFileDao" ref="kfxFileDao" />
</bean>
```

We specify the unique identifier by how we can reference the service in the Spring configuration file and the implementation class for this service. The service requires access to a KFX File DAO so we inject a reference to the earlier configured DAO into the service.

As you can see now we can easily change to a different ORM tool or database implementation by changing the implementation class of the kfxFileDao bean in the Spring configuration file. No other code changes would be required in our Java code. Spring would just inject the other implementation of the DAO and the service would not know that it is now using a different ORM tool. This is one example of the benefits of Spring.

4.6 Transactions

When writing to a database, we must ensure that the integrity of the data is maintained by performing the updates within a transaction. In the grand tradition of software development, an acronym has been created to describe transactions: ACID. In short ACID stands for:

- *Atomic* - Transactions are made up of one or more activities bundled together as a single unit of work. Atomicity ensures that all the operations in the transaction happen or that none of them happen. If all the activities succeed, the transaction is a success. If any of the activities fail, the entire transaction fails and is rolled back.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

- *Consistent* - Once a transaction ends (whether successful or not), the system is left in a state that is consistent with the business that it models. The data should not be corrupted with respect to reality.
- *Isolated* - Transactions should allow multiple users to work with the same data, without each user's work getting tangled up with the others. Therefore, transactions should be isolated from each other, preventing concurrent reads and writes to the same data from occurring. (Note that isolation typically involves locking rows and/or tables in a database.)
- *Durable* - Once the transaction has completed, the results of the transaction should be made permanent so that they will survive any sort of system crash. This typically involves storing the results in a database or some other form of persistent storage.

Spring has rich support for transaction management, both programmatic and declarative. Since we like to keep our code as clean as possible with just the business logic we focus here on only using declarative transactions. Declarative transactions help us to decouple an operation from its transaction rules. The transactions are completely configured in the Spring configuration files and no coding is required at all to add the transaction logic. The only limitation with using Spring declarative transactions is that they can only be applied to methods. If for any reason a transaction requires control at a level within a method programmatic transactions need to be used.

4.6.1 Transaction Manager

Spring does not directly manage transactions. Instead, it comes with a selection of transaction managers that delegate responsibility for transaction management to a platform-specific transaction implementation provided by either JTA or the persistence mechanism. Since we use Hibernate 3 for our persistence layer we need to use the orm.hibernate3.HibernateTransactionManager. When using JDBC or iBATIS you would use the jdbc.datasource.DataSourceTransactionManager. When using the Java Persistence API (JPA) you would need to use the orm.jpa.JpaTransactionManager. More transaction managers are available for Toplink, J2EE connector, JMS, etc.

To configure the transaction manager add the following configuration to the applicationContext.xml files:

```
<!--     TRANSACTION MANAGEMENT CONFIGURATION          -->

<bean id="txManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

The `sessionFactory` property should be wired with the Hibernate `SessionFactory` we configured earlier and which has as bean identifier `sessionFactory` and is referenced as such here.

`HibernateTransactionManager` delegates responsibility for transaction management to an `org.hibernate.Transaction` object that it retrieves from the Hibernate session. When a transaction successfully completes, `HibernateTransactionManager` will call the `commit()`

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

method on the `Transaction` object. Similarly, when a transaction fails, the `rollback()` method will be called on the `Transaction` object.

4.6.2 Transaction Attributes

In Spring, declarative transactions are defined with transaction attributes. A transaction attribute is a description of how transaction policies should be applied to a method. There are five facets of a transaction attribute:

- Propagation
- Isolation
- Read-only?
- Timeout
- Rollback Rules

Propagation behavior

The first facet of a transaction is propagation behavior. Propagation behavior defines the boundaries of the transaction with respect to the client and to the method being called. Spring defines seven distinct propagation behaviors.

- PROPAGATION_MANDATORY; Indicates that the method must run within a transaction. If no existing transaction is in progress, an exception will be thrown.
- PROPAGATION_NESTED; Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rolled back individually from the enclosing transaction. If no enclosing transaction exists, behaves like PROPAGATION_REQUIRED. Vendor support for this propagation behavior is spotty at best. Consult the documentation for your resource manager to determine if nested transactions are supported. Note that Hibernate does not support nested transactions.
- PROPAGATION_NEVER; Indicates that the current method should not run within a transactional context. If there is an existing transaction in progress, an exception will be thrown.
- PROPAGATION_NOT_SUPPORTED; Indicates that the method should not run within a transaction. If an existing transaction is in progress, it will be suspended for the duration of the method. If using `JTATransactionManager`, access to `TransactionManager` is required.
- PROPAGATION_REQUIRED; Indicates that the current method must run within a transaction. If an existing transaction is in progress, the method will run within that transaction. Otherwise a new transaction will be started.
- PROPAGATION_REQUIRES_NEW; Indicates that the current method must run within its own transaction. A new transaction is started and if an existing transaction is in progress, it will be suspended for the duration of the method. If using `JTATransactionManager`, access to `TransactionManager` is required.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

- PROPAGATION_SUPPORTS; Indicates that the current method does not require a transactional context, but may run within a transaction if one is already in progress.

Propagation rules answer the question of whether a new transaction should be started or suspended, or if a method should even be executed within a transactional context at all.

Isolation Levels

The second dimension of a declared transaction is the isolation level. An isolation level defines how much a transaction may be impacted by the activities of other concurrent transactions. Another way to look at a transaction's isolation level is to think of it as how selfish the transaction is with the transactional data.

In a typical application, multiple transactions run concurrently, often working with the same data to get their job done. Concurrency, while necessary, can lead to the following problems:

- *Dirty Read*: Dirty reads occur when one transaction reads data that has been written but not yet committed by another transaction. If the changes are later rolled back, the data obtained by the first transaction will be invalid.
- *Nonrepeatable Read*: Nonrepeatable reads happen when a transaction performs the same query two or more times and each time the data is different. This is usually due to another concurrent transaction updating the data between queries.
- *Phantom Reads*: Phantom reads are similar to nonrepeatable reads. These occur when a transaction (T1) reads several rows, and then a concurrent transaction (T2) inserts rows. Upon subsequent queries the first transaction (T1) finds additional rows that were not there before.

The level of isolation chosen for a transaction can affect the performance of an application. Isolation often involves locking rows and sometimes complete tables in the data store. Aggressive locking can hinder concurrency, requiring transactions to wait on each other to do their work.

The following levels of isolation are possible in Spring:

- ISOLATION_DEFAULT; Use the default isolation level of the underlying data store. The default for the InnoDB configured MySQL database is ISOLATION_REPEATABLE_READ.
- ISOLATION_READ_UNCOMMITTED; Allows you to read changes that have not yet been committed. May result in dirty reads, phantom reads, and nonrepeateable reads.
- ISOLATION_READ_COMMITTED; Allows reads from concurrent transactions that have been committed. Dirty reads are prevented, but phantom and nonrepeateable reads may still occur.
- ISOLATION_REPEATABLE_READ; Multiple reads of the same field will yield the same results, unless changed by the transaction itself. Dirty reads and nonrepeateable reads are prevented, but phantom reads may still occur.
- ISOLATION_SERIALIZABLE; The fully ACID-compliant isolation level ensures that dirty reads, nonrepeateable reads, and phantom reads are all prevented. This is the slowest of all isolation

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

levels because it is typically accomplished by doing full table locks on the tables involved in the transaction.

ISOLATION_READ_UNCOMMITTED is the most efficient isolation level, but isolates the transaction the least. ISOLATION_SERIALIZABLE prevents all forms of isolation problems but is the least efficient.

Not all isolation levels are supported by all data sources. Our configured data source using the MySQL driver and InnoDB tables supports all four isolation levels.

Read-Only

The third characteristic of a declared transaction is whether it is a read-only transaction. If a transaction performs only read operations against the underlying data store, the data store may be able to apply certain optimizations that take advantage of the read-only nature of the transaction. By declaring a transaction as read-only you give the underlying data store the opportunity to apply those optimizations as it sees fit.

Because read-only optimizations are applied by the underlying data store when a transaction begins, it only makes sense to declare a transaction as read-only on methods with propagation behaviors that may start a new transaction (PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW, and PROPAGATION_NESTED).

Furthermore, when using Hibernate as the persistence mechanism, declaring a transaction as read-only will result in Hibernate's flush mode being set to FLUSH_NEVER. This tells Hibernate to avoid unnecessary synchronization of objects with the database, thus delaying all updates until the end of the transaction.

Transaction Timeout

Fir an application to perform well, its transactions can't carry on for a long time. Therefore, the next trait of a declared transaction is its timeout. The timeout is specified in seconds. The timeout clock begins when a transaction starts, so it only makes sense to declare a timeout on methods with propagation behaviors that may start a new transaction (PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW, and PROPAGATION_NESTED).

Rollback Rules

The final facet of the transaction pentagon is a set of rules that define what exceptions prompt a rollback and which ones do not. By default, transactions are rolled back only on runtime exceptions and not on checked exceptions. (This behavior is consistent with rollback behavior in EJBs.)

You can declare that a transaction be rolled back on specific checked exceptions as well as runtime exceptions. Likewise, you can declare that a transaction not roll back on specified exceptions, even if those exceptions are runtime exceptions.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

4.6.3 Declaring Transactions

The configuration elements for declaring transactions are defined in the Spring tx namespace. Note that the aop namespace is also required since the transaction elements rely on a few of Spring's AOP configuration elements. The transaction are defined using AspectJ aspects.

First we need to configure the transaction aspect's advice specifying which transaction attributes to apply to what methods. Add the following to the applicationContext.xml files.

```
<!--  The transaction advice for the kfxFileService -->
<tx:advice id="txKfxFileServiceAdvice" transaction-manager="txManager">
  <tx:attributes>
    <!-- all methods starting with 'add' have the properties:
         propagation-required - method must run in a transaction if a tx is
         in progress it will run in that transaction
         isolation-level - default -->
    <tx:method name="add*"
            propagation="REQUIRED" />

    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="get*"
            propagation="REQUIRED"
            isolation="REPEATABLE_READ"
            read-only="true" />

    <!-- all other methods can run in a transaction if needed, but
         don't require it -->
    <tx:method name="*"
            propagation="SUPPORTS"
            read-only="true" />
  </tx:attributes>
</tx:advice>
```

With the advice we need to specify which transaction manager to use. In our case we named our transaction manager 'txManager'. If we would have named our transaction manager 'transactionManager' we would not have had to declare the transaction-manager attribute since it would use a transaction manager by that name by default.

The <tx:method>'s name attribute is used to define to which method(s) the attributes need to be applied. Wildcards can be used to apply the attributes to multiple methods. In our case we advice three sets of methods, all add methods (`addKfxFile`), all get methods (`getKfxFile, getKfxFiles`), and any other remaining methods (none currently in our KFX File Service).

For each <tx:method> we can declare a set of attributes to define the method's transaction policies. The following properties can be specified:

- propagation; The transaction's propagation rule
- isolation; The transaction isolation level
- read-only; To specify whether a transaction is read-only
- timeout; the timeout in seconds for a long-running transaction

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

- rollback-for; the comma-delimited list of qualified class name for the checked exceptions for which a transaction should be rolled back and not committed
- no-rollback-for; the comma-delimited list of qualified class name for the exceptions for which a transaction should continue and not be rolled back

For our application we defined that all 'add' methods require the PROPAGATION_REQUIRED propagation behavior and use the ISOLATION_DEFAULT isolation level (default when not specified), it is not read-only, no timeout is specified, and no rollback rules are specified.

The 'get' methods require the PROPAGATION_REQUIRED propagation behavior, ISOLATION_REPEATABLE_READ isolation level, are read-only and require no timeout or rollback rules.

All remaining methods support support transactions and are defined to be read-only using as default isolation level ISOLATION_DEFAULT with no timeout and no rollback rules.

On its own the <tx:advice> only defines an AOP advice for advising methods with transaction boundaries. But this is only transaction advice, not a complete transactional aspect. Nowhere in <tx:advice> did we indicate which beans should be advised - we need a pointcut for that. To completely define the transaction aspect, we must define an advisor. The following configuration defines an advisor that uses the txKfxFileServiceAdvice advice to advise any beans that implement the IKfxFileService interface. Add the following to the applicationContext.xml files.

```
<!--  Apply the transaction advice to all IKfxFileService implementations -->
<aop:config>
  <aop:advisor
    pointcut="execution(* gov.nasa.arc.ci.spring.common.IKfxFileService.*(..))"
    advice-ref="txKfxFileServiceAdvice" />
</aop:config>
```

The pointcut attribute uses an AspectJ pointcut expression to indicate that this advisor should advise all methods of the IKfxFileService interface. Which methods are actually run within a transaction and what the transactional attributes are for those methods is defined by the transaction advise, which is referenced with the advice-ref attribute to be the advise named txKfxFileServiceAdvice.

The pointcut expression indicates that the pointcut is to be applied during the execution of all methods in the IKfxFileService with any return type (* gov...), with any method name (IKfxFileService.*) with any arguments ((..)).

Spring also allows you to declare your transactions using annotations in your class files. For our application we chose not to use annotations. The problem with annotations is that if you wish to change any of the transaction properties you will need to modify the Java code, recompile the code, and repackage and redeploy the compiled code. Configuring the transactions declaratively in the configuration file allows you to change the transaction configuration in Spring's configuration file without having to modify the Java classes.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

This takes care of the transaction boundaries for the methods in the KFX File Service and completes the creation and configuration of all the data services.

5 Application Services

This simple application uses two application services each to reside on a different system and therefore managed by different actors hosted in different actor hosting environments. One of the application services is the User Interface Application Service responsible for the display of the user interface and managing the interactions with the user. The other application is the KFX Parser responsible for parsing KFX log files and providing the parsed KFX File data.

5.1 User Interface Application Service

The User Interface Application Service is the service responsible for displaying the user interface to the user and managing user interactions. For this service we defined the contract in the IUIApplication interface and its implementation in the UIApplication class.

5.1.1 IUIApplication Interface

The IUIApplication interface is the contract for the user interface application service and is defined as:

```
package gov.nasa.arc.ci.spring.ui;
public interface IUIApplication {
  /**
   * Initializes the user interface.
   */
  public void initialize();

  /**
   * Displays or hides the application frame.
   *
   * @param visible true to visualize the application, false to hide it
   */
  public void setVisible(boolean visible);

  /**
   * Exits the application.
   */
  public void onExit();
} // IUIApplication
```

In this contract we specify that we need to be able to initialize the user interface have the ability to control the visibility of the user interface and be able to exit the user interface closing the application and performing any application cleanup.

5.1.2 UIApplication Class

The UIApplication Class provides the implementation of the user interface application service. It specifies what is displayed as part of the user interface and the operation performed when the user interacts with the user interface.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

The application is a Swing JFrame and abides by the contract for the user interface application service.

```
package gov.nasa.arc.ci.spring.ui;

import gov.nasa.arc.ci.spring.actor.gui.IGuiActor;
import gov.nasa.arc.ci.spring.common.IKfxFileService;
import gov.nasa.arc.ci.spring.common.KfxFile;
import gov.nasa.ci.api.communication.CommunicativeActFactory;
import gov.nasa.ci.api.communication.ICommunicativeAct;
import gov.nasa.ci.api.communication.IPayload;
import gov.nasa.ci.common.actor.communication.AbstractCIReplyHandler;
import gov.nasa.ci.common.actor.directory.AbstractCIActorLookupListener;
import gov.nasa.ci.common.actor.directory.CIActorLookupMethod;
import gov.nasa.ci.corba.api.Credentials;
import gov.nasa.ci.corba.api.communication.CAPerformative;
import gov.nasa.ci.corba.api.communication.ICAPayloadProperty;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Set;

import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;

import org.apache.log4j.Logger;

public class UIApplication extends JFrame implements IUIApplication {

  private final static Logger LOGGER = Logger.getLogger(UIApplication.class);
```

The LOGGER is a Log4J logger used to log informational messages into the log file managed by the CI.

The constructor for the service is a pretty straightforward constructor that sets the title for the frame, sets the default operation for the close button in the frame, and specifies the initial size of the frame.

```
public UIApplication() {
  super("CI Spring Application");
  setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
  setSize(800, 400);
} // UIApplication
```

As part of the initialization of the user interface we create the parse button to allow the user to initiate the parsing of KFX logs, and a list view to display the parsed results. The parse button is by default disabled and is only enabled when the actor providing the parsing service has been located via the directory service.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
/** The KFX Parser Actor */
private Credentials       m_oKFXActor;

/** The button used to initiate KFX log parsing */
private JButton           m_oParseButton;
/** The list with parsed KFX files */
private JList             m_oKFXFilesView;
/** The list model managing the files displayed in the files view */
private DefaultListModel  m_oKFXFilesListModel;

public void initialize() {
  // create UI components
  m_oParseButton = new JButton("Parse");
  m_oParseButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      onParseRequest();
    } // actionPerformed
  });
  m_oParseButton.setEnabled(false);

  m_oKFXFilesView = new JList(m_oKFXFilesListModel = new DefaultListModel());
  JScrollPane oFilesScrollPane = new JScrollPane();
  oFilesScrollPane.getViewport().setView(m_oKFXFilesView);

  // layout UI components
  getContentPane().setLayout(new BorderLayout());
  getContentPane().add(m_oParseButton, BorderLayout.NORTH);
  getContentPane().add(oFilesScrollPane, BorderLayout.CENTER);


  // get reference to remote KFX Parser actor
  getActor().lookupActor(
      "gov.nasa.arc.ci.spring.actor.parser.KfxParserActor",
      0,                                 // keep trying to locate the actor
      CIActorLookupMethod.DIRECTORY,     // lookups via the directory service
      new AbstractCIActorLookupListener(getActor()) {
        @Override
        public void onActorFound(Credentials actor) {
          m_oKFXActor = actor;
          SwingUtilities.invokeLater(new Runnable() {
            public void run() {
              m_oParseButton.setEnabled(true);
            } // run
          });
        } // onActorFound
      });
} // initialize
```

When the parse button is pressed the onParseRequest method is invoked. The list view is placed in a scroll pane to allow the user to scroll through all parsed files when the number of files exceeds the number of files that fit in the view on the screen. This code should look familiar to anyone that coded with Swing components.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

The last portion of the initialization method is more interesting. Here we use the managing actor's lookup method to locate the KFX Parser Actor. The UIApplication is made aware of the actor that manages this application via the Spring dependency injection. To allow Spring to inject a reference to the actor we need to add the appropriate setter method. We also added the appropriate getter, not required for Spring.

```
private IGuiActor         m_oActor;

public void setActor(IGuiActor actor) {
  m_oActor = actor;
} // setActor

public IGuiActor getActor() {
  return m_oActor;
} // getActor
```

To get back to the actor lookup. The lookupActor method requires us to specify the name of the actor to lookup. We use directory based lookup and therefore need to provide the directory name by which the actor is registered in the directory service. We can also provide a timeout in milliseconds for how long we want to try to locate the actor before giving up, in this case we do not ever want to give up. The lookup can use two approaches, one is directory based (pull), the other heartbeat based (push). With the directory based lookup the CI periodically polls the directory service to see if an actor by the specified name has been registered. With the heartbased lookup the CI registers a heartbeat subscriber with the actor's management service and waits until it receives a heartbeat for the actor. In case of a heartbeat based lookup the actor's globally unique qualified name must be provided, not the directory name. For heartbeat based lookup to work the actor must also be configured to publish its heartbeat. This was covered earlier in this document. This lookupActor method uses an asynchronous notification which is what we want. Other lookup methods are available, some which are synchronous blocking until the lookup succeeds or times out. For the asynchronous lookup we need to provide a lookup listener. The CI provides a convenient AbstractCILookupListener. Since we don't use a timeout the only method to extend is the onActorFound method which notifies the listener of the Credentials of the actor. When invoked we cache those Credentials and enable the parse button. Note that the listener's method is invoked on a CI managed thread and not the AWT event queue so the enabling of the parse button must be invoked on the AWT event queue as per the AWT/Swing development guidelines. The following specifies the field and accessor method for the KFX actor.

```
private Credentials       m_oKFXActor;
public Credentials getKFXActor() {
  return m_oKFXActor;
} // getKFXActor
```

When the user presses the parse button we need to send a request to the KFX Parser actor requesting it to parse the log files and to send us back the results. This is handled in the onParseRequest method.

```
/**
 * Requests files to be parsed from the KFX log files.
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
 */
public void onParseRequest() {
  m_oParseButton.setEnabled(false);
  ICommunicativeAct oRequest = CommunicativeActFactory.createRequest(
      getActor().getCredentials(), getKFXActor(),
      "parse");
  try {
    getActor().sendAsynchronousMessage(
        oRequest,
        new AbstractCIReplyHandler(getActor(), oRequest, 30000, true) {
          @Override
          public void onReply(
              ICommunicativeAct reply,
              ICommunicativeAct inReplyTo) {
            IPayload oPayload = reply.getPayload();
            CAPerformative oPerformative = oPayload.getPerformative();
            switch (oPerformative.value()) {
              case CAPerformative._INFORM: {
                // files parsed, display files in file view
                Set<String> coKeys =
                    (Set<String>)oPayload.getSerializablePropertyValue(
                          ICAPayloadProperty.CONTENT);
                try {
                  final Set<KfxFile> coFiles =
                          m_oKfxFileService.getKfxFiles(coKeys);
                  SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                      for (KfxFile oKfxFile : coFiles) {
                        LOGGER.info("Adding file: "+oKfxFile.getKey());
                        m_oKFXFilesListModel.addElement(oKfxFile.getKey());
                      } // end for
                    } // run
                  });
                } catch (Throwable tx) {
                  LOGGER.error(
                          "Failed to retrieve KfxFiles from database.",
                          tx);
                } // end try
                break;
              } // INFORM
              case CAPerformative._FAILURE: {
                // failure parsing files
                String sReason = oPayload.getStringPropertyValue("reason");
                String sMessage = oPayload.getStringPropertyValue("message");
                LOGGER.error(
                    "KFX Actor reported failure parsing KFX logs; reason: "+
                    sReason+", message: "+sMessage);
                break;
              } // FAILURE
            } // end switch

            // re-enable parse button
            SwingUtilities.invokeLater(new Runnable() {
              public void run() {
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
                    m_oParseButton.setEnabled(true);
                  } // run
                });
              } // onReply
              @Override
              public void onTimeOut() {
                LOGGER.warn(
                        "Timed out waiting for a response for the parse request.");
                SwingUtilities.invokeLater(new Runnable() {
                  public void run() {
                    m_oParseButton.setEnabled(true);
                  } // run
                });
              } // onTimeOut
            });
      } catch (Throwable tx) {
        LOGGER.error("Failed to send parse request.", tx);
        m_oParseButton.setEnabled(true);
      } // end try
    } // onParseRequest
```

When the method is invoked we first disable the parse button, since we don't want another request to be sent while we are processing a request to do so. This method is invoked on the AWT event queue, so no need to have it explicitly invoked on that queue.

Next we need to create the request CI CommunicativeAct specifying the sender, recipient, and the action (parse). No other information is required in the payload for this request. This is all required as part of the messaging contract for the KFX parser actor.

Next we send the request asynchronously to the KFX parser actor. We send it asynchronously so that we won't block the AWT event queue and allow for the user to perform other actions in the user interface. As part of the request CommunicativeAct a conversation identifier was generated for us. We can track responses for this request using this conversation identifier. To make use of this conversation tracking we can register a reply handler with the request to have the message sent. The CI will in that case make sure that the reply handler is notified of all responses for the specified request. There will be no need in that case to have to separately register message handlers for the actor and to manually track those conversations. In our case we use the CI provided AbstractCIReplyHandler which provides configuration options to time out after an amount of time when no response is received and to automatically deregister the reply handler with the CI when a response is received or the time out timer expires. The first argument for this reply handler is a reference to the requesting actor, needed to use its transport service to deregister the reply handler if desired. The second argument is the request for which responses are expected. The third argument the timeout in milliseconds, 30 seconds in this case, and the last argument specifies whether to automatically deregister the reply handler or not. When true it will deregister on the first incoming response. If multiple response messages were expected this should be set to false and the developer would need to take of deregistering the reply handler. The sendAsynchronousMessage returns an identifier assigned to the reply handler that needs to be used to deregister the reply handler.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

When we receive an INFORM response notifying us of the parsed KFX file identifiers/keys we want to obtain the KFX Files from the database using those keys. For this we need access to the KFX File Service. Of course we want Spring to inject this. So we need to provide the setter for this service.

```
private IKfxFileService   m_oKfxFileService;
public void setKfxFileService(IKfxFileService service) {
  m_oKfxFileService = service;
} // setKfxFileService
```

When we receive the response we need to extract the keys from the message's payload:

```
// files parsed, display files in file view
Set<String> coKeys = (Set<String>)oPayload.getSerializablePropertyValue(
             ICAPayloadProperty.CONTENT);
```

Next we use the KFX File Service to retrieve the files from the database:

```
final Set<KfxFile> coFiles = m_oKfxFileService.getKfxFiles(coKeys);
```

Last we want to display the files in the view. In this simple application we just display the keys, so in theory we would not have needed to retrieve the files from the database, but imagine a more advanced table that displays the various properties of the file.

```
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    for (KfxFile oKfxFile : coFiles) {
      LOGGER.info("Adding file: "+oKfxFile.getKey());
      m_oKFXFilesListModel.addElement(oKfxFile.getKey());
    } // end for
  } // run
});
```

Since the display of data is a user interface operation we again need to make sure that this happens on the AWT Event Queue which is why we wrap the display in the invokeLater call.

That takes care of the roundtrip implementation of requesting files to be parsed and displaying those files.

5.1.3 Spring Configuration

The application service is a Spring managed bean and as mentioned in the previous section the application service depends on both the actor managing the application and the file service so we need to configure this application in the Spring configuration file. We need to add the following to mas-applicationContext.xml.

```
<bean id="application"
    class="gov.nasa.arc.ci.spring.ui.UIApplication">
  <property name="actor" ref="guiActor" />
  <property name="kfxFileService" ref="kfxFileService" />
</bean>
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

We provide the implementation class to use for the application and provide references to the guiActor bean (which remember is the proxy to the actual actor since abstract beans cannot be injected into other beans) and a reference to the KFX File Service. The application implementation is not aware that it is invoking methods on the proxy and not the actual actor because of our use of the IGuiActor interface as the contract for the actor.

5.2 KFX Parser Service

The KFX Parser Service provides the service for parsing KFX log files and returning references to the parsed data back to the requestor. For this service we defined the contract in the IKfxParser interface and its implementation in the KfxParser class.

5.2.1 IKfxParser Interface

The contract for the KFX Parser is very simple. It consists of one method to initiate the parsing and for it to return the set of parsed KfxFiles.

```
package gov.nasa.arc.ci.spring.parser;

import gov.nasa.arc.ci.spring.common.KfxFile;

import java.util.Set;

public interface IKfxParser {

  /**
   * Parses KFX logs and returns the set of parsed KFXFiles.
   *
   * @return the set of parsed KFX files.
   */
  public Set<KfxFile> parse();

} // IKfxParser
```

5.2.2 KfxParser Class

The implementation of the KFX Parser is also very straightforward. To keep the application simple there is no parsing of files involved. All we do is create five new KfxFile objects at a time. We maintain a line number to ensure uniqueness of the files. The one interesting thing is that we also store each new 'parsed' KFX file in the data store. We do this one single line of code. It does mean that this parser requires access to the KFX File Service to allow us to persist the KFX files. Again this association is managed via the Spring configuration file.

```
package gov.nasa.arc.ci.spring.parser;

import gov.nasa.arc.ci.spring.common.Action;
import gov.nasa.arc.ci.spring.common.IKfxFileService;
import gov.nasa.arc.ci.spring.common.KfxFile;

import java.util.Date;
import java.util.HashSet;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
import java.util.Set;

public class KfxParser implements IKfxParser {

  /** The IKfxFileService to use to store newly parsed files */
  private IKfxFileService   m_oKfxFileService;
  /** The last line number at which a KfxFile was parsed */
  private int               m_nLineNumber = 1;

  public void setKfxFileService(IKfxFileService service) {
    m_oKfxFileService = service;
  } // setKfxFileService

  public Set<KfxFile> parse() {
    Set<KfxFile> coFiles = new HashSet<KfxFile>();
    for (int i = 0; i < 5; i++) {
      KfxFile oKfxFile = new KfxFile(
          "PlanB",
          m_nLineNumber++,
          Action.Uplink,
          new Date(),
          "File_"+(m_nLineNumber-1)+".txt",
          "txt",
          24875,
          "C:/oca--data/oca-up");
      m_oKfxFileService.addKfxFile(oKfxFile);
      coFiles.add(oKfxFile);
    } // end for
    return coFiles;
  } // parse

} // KfxParser
```

Adding a file to the data store is performed with the one line of code:

```
m_oKfxFileService.addKfxFile(oKfxFile);
```

5.2.3 Spring Configuration

The parser service is a Spring managed bean and as mentioned in the previous section the service depends on the file service so we need to configure this application in the Spring configuration file. We need to add the following to oca-applicationContext.xml.

```
<!--            PARSER CONFIGURATION                  -->

<bean id="kfxParser"
    class="gov.nasa.arc.ci.spring.parser.KfxParser">
  <property name="kfxFileService" ref="kfxFileService" />
</bean>
```

We provide the implementation class to use for the parser and provide a reference to the KFX File Service.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

5.3 CI Message Handling

So far we showed how the UI application service can send a message to the parser actor and how the parser service implemented, but missing is how the request message is processed by the KFX parser actor and how as part of that processing the parser is used.

Every actor requires an ICIMessageHandlerFactory to be set which is used to provide an appropriate message handler for every CommunicativeAct sent to that actor. The factory can examine the content of the message and based on the content determine which ICIMessageHandler handler is most appropriate to process the CommunicativeAct. For our application we developed a generic CIMessageHandlerFactory that chooses an appropriate message handler based on the payload's performative and action values. It requires message handlers to implement the ICIActionMessageHandler Interface.

In our application we developed a ParseMessageHandler that is responsible for processing CommunicativeActs with as performative REQUEST and as action 'parse'. It is this ParseMessageHandler that interacts with the parser to have the KFX log files parsed.

5.3.1 CIMessageHandlerFactory Class

The CIMessageHandlerFactory is responsible for providing an ICIMessageHandler depending on the performative and action of a CommunicativeAct. To make this factory configurable in Spring we allow for the injection of message handlers that implement the ICIActionMessageHandler interface.

```
package gov.nasa.arc.ci.spring.common.communication;

import java.util.HashMap;
import java.util.Set;

import gov.nasa.ci.api.communication.ICommunicativeAct;
import gov.nasa.ci.common.actor.communication.ICIMessageHandler;
import gov.nasa.ci.common.actor.communication.ICIMessageHandlerFactory;
import gov.nasa.ci.corba.api.communication.CAPerformative;

public class CIMessageHandlerFactory implements ICIMessageHandlerFactory {

  public CIMessageHandlerFactory() {
    m_moHandlers = new HashMap<Integer, HashMap<String, ICIMessageHandler>>();
  } // CIMessageHandlerFactory

  /** The list of ICIMessageHandlers by performative and action */
  private HashMap<Integer, HashMap<String, ICIMessageHandler>> m_moHandlers;

  /**
   * Registers the specified message handlers with this factory
   * for handling messages with the specified performative
   * and action.
   *
   * @param handlers the message handlers
   */
  public void setHandlers(Set<ICIActionMessageHandler> handlers) {
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    for (ICIActionMessageHandler oHandler : handlers) {
      addHandler(oHandler.getPerformative(), oHandler.getAction(), oHandler);
    } // end for
  } // setHandlers

  /**
   * Adds the specified ICIMessageHandler to this factory registered
   * for handling messages with the specified performative and action.
   *
   * @param performative the CAPerformative for the message to be processed
   * @param action the action value for the message to be processed
   * @param handler the ICIMessageHandler to add
   */
  protected void addHandler(
       CAPerformative performative,
       String action,
       ICIMessageHandler handler) {
    HashMap<String, ICIMessageHandler> moHandlers =
             m_moHandlers.get(performative.value());
    if (moHandlers == null) {
      moHandlers = new HashMap<String, ICIMessageHandler>();
      m_moHandlers.put(performative.value(), moHandlers);
    } // end if
    moHandlers.put(action, handler);
  } // addHandler

  /**
   * Returns a ICIMessageHandler for the specified ICommunicativeAct.
   *
   * @param message the ICommunicativeAct for which to return a handler
   * @return ICIMessageHandler the handler for the specified message
   */
  public ICIMessageHandler getHandler(ICommunicativeAct message) {
    ICIMessageHandler oHandler = null;
    CAPerformative oPerformative = message.getPayload().getPerformative();
    if (oPerformative != null) {
      HashMap<String, ICIMessageHandler> moHandlers =
             m_moHandlers.get(oPerformative.value());
      if (moHandlers != null) {
        String sAction = message.getPayload().getAction();
        if (sAction != null) {
          oHandler = moHandlers.get(sAction);
        } // end if
      } // end if
    } // end if
    return oHandler;
  } // getHandler

} // CIMessageHandlerFactory
```

The injection point for message handlers is the setHandlers method. It allows for multiple handlers to be injected by providing the factory a set of ICIActionMessageHandlers. In the Spring Configuration section we will show how to register these message handlers.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

5.3.2 ICIActionMessageHandler Interface

Since our factory determines the appropriate handler by performative and action we defined a contract for the message handlers for that factory to provide the performative and action for which they can process messages.

```
package gov.nasa.arc.ci.spring.common.communication;

import gov.nasa.ci.common.actor.communication.ICIMessageHandler;
import gov.nasa.ci.corba.api.communication.CAPerformative;

/**
 * ICIActionMessageHandler is an interface for a handler used to
 * process Collaborative Infrastructure ICommunicativeActs
 * for a specific performative and action defined in the
 * payload of ICommunicativeActs.
*/
public interface ICIActionMessageHandler extends ICIMessageHandler {

  /**
   * Returns the performative that must be defined in the ICommunicativeActs
   * to be processed by this message handler.
   *
   * @return CAPerformative the performative
   */
  public CAPerformative getPerformative();

  /**
   * Returns the action that must be defined in the ICommunicativeActs
   * to be processed by this message handler.
   *
   * @return String the action
   */
  public String getAction();

} // ICIActionMessageHandler
```

5.3.3 ParseMessageHandler Class

Since our KFX Parser Actor needs to be able to respond to parse requests we developed a message handler that takes care of processing the parse request CommunicativeActs.

```
package gov.nasa.arc.ci.spring.actor.parser.communication;

import java.util.HashSet;
import java.util.Set;

import gov.nasa.arc.ci.spring.actor.parser.IKfxParserActor;
import gov.nasa.arc.ci.spring.common.KfxFile;
import gov.nasa.arc.ci.spring.parser.IKfxParser;
import gov.nasa.ci.api.communication.CommunicativeActFactory;
import gov.nasa.ci.api.communication.ICommunicativeAct;
import gov.nasa.ci.common.actor.IActorTask;
import gov.nasa.ci.corba.api.AnyFactory;
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
import gov.nasa.ci.corba.api.communication.CAPerformative;

import org.apache.log4j.Logger;

public class ParseMessageHandler extends AbstractCIMessageHandler {

  /** The action for the messages handled by this handler */
  public final static String ACTION = "parse";

  /** Logger used to log messages. */
  private final static Logger LOGGER =
             Logger.getLogger(ParseMessageHandler.class);

  /**
   * Constructor, creates a new ParseMessageHandler that processes
   * messages for a IKfxParserActor.
   */
  public ParseMessageHandler() {
  } // ParseMessageHandler

  /**
   * Constructor, creates a new ParseMessageHandler that processes
   * messages for the specified IKfxParserActor.
   *
   * @param actor the IKfxParserActor for which this handler processes messages
   */
  public ParseMessageHandler(IKfxParserActor actor) {
    super(actor);
  } // ParseMessageHandler

  /** The parser used to parse KFX logs */
  private IKfxParser m_oParser;

  /**
   * Sets the parser to use to parse KFX logs.
   *
   * @param parser the IKfxParser
   */
  public void setParser(IKfxParser parser) {
    m_oParser = parser;
  } // setParser

  public CAPerformative getPerformative() {
    return CAPerformative.REQUEST;
  } // getPerformative

  public String getAction() {
    return ACTION;
  } // getAction

  /**
   * Processes the specified ICommunicativeAct.
   *
   * @param message the ICommunicativeAct to process
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
     */
    public void process(final ICommunicativeAct message) {
      getActor().postActorTask(new IActorTask(){
        public void run() {
          Set<KfxFile> coKFXFiles = m_oParser.parse();
          Set<String> coKFXFileKeys = new HashSet<String>();
          for (KfxFile oKFXFile : coKFXFiles) {
            coKFXFileKeys.add(oKFXFile.getKey());
          } // end for

          // return response
          try {
            ICommunicativeAct oResponse = CommunicativeActFactory.createResponse(
                getActor().getCredentials(),
                CAPerformative.INFORM,
                AnyFactory.create(coKFXFileKeys),
                message);
            getActor().sendMessage(oResponse);
          } catch (Throwable tx) {
            LOGGER.error("Failed to return response with parse results.", tx);
          } // end try
        } // run
      });
    } // process

  } // ParseMessageHandler
```

This message handler needs to be able to make use of the CI and actor services of the actor for which this message handler processed messages. We therefore need to be able to associate the actor with this message handler. For that purpose we developed the AbstractCIMessageHandler which provides a method to set the actor for the message handlers serving the parser actor. Since this message handler needs to make use of the parsing services offered by the KFX Parser we add a setter method for the parser to allow us to inject the parser.

As part of the implementation of the actual processing of the CommunicativeAct we have the actor do the actual processing on its worker thread by posting an IActorTask on its work queue for processing.

```
getActor().postActorTask(new IActorTask(){
```

It then asks the parser to parse files and stores the keys of the files in a Set to be returned to the requesting actor.

```
Set<KfxFile> coKFXFiles = m_oParser.parse();
Set<String> coKFXFileKeys = new HashSet<String>();
for (KfxFile oKFXFile : coKFXFiles) {
  coKFXFileKeys.add(oKFXFile.getKey());
} // end for
```

Last we create a response CommunicativeAct and provide the results as the content for the message and send the message to the requestor.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
ICommunicativeAct oResponse = CommunicativeActFactory.createResponse(
              getActor().getCredentials(),
              CAPerformative.INFORM,
              AnyFactory.create(coKFXFileKeys),
              message);
getActor().sendMessage(oResponse);
```

The CommunicativeActFactory takes care of setting the conversation identifier of the request message in the response. It also obtains the sender of the request message and set it as the recipient for the response message. We use the AnyFactory to wrap the keys into a CORBA Any required as the value for the content property in the payload of the CommunicativeAct.

The AbstractCIMessageHandler is implemented as follows:

```
package gov.nasa.arc.ci.spring.actor.parser.communication;

import gov.nasa.arc.ci.spring.actor.parser.IKfxParserActor;
import gov.nasa.arc.ci.spring.common.communication.ICIActionMessageHandler;
import gov.nasa.ci.api.communication.ICommunicativeAct;
import gov.nasa.ci.corba.api.communication.CAPerformative;

public abstract class AbstractCIMessageHandler
       implements ICIActionMessageHandler {

  /**
   * Constructor, creates a new AbstractCIMessageHandler that processes
   * messages for a KfxParserActor.
   */
  public AbstractCIMessageHandler() {
  } // AbstractCIMessageHandler

  /**
   * Constructor, creates a new AbstractCIMessageHandler that processes
   * messages for the specified IKfxParserActor.
   *
   * @param actor the IKfxParserActor for which this handler processes messages
   */
  public AbstractCIMessageHandler(IKfxParserActor actor) {
    m_oActor = actor;
  } // AbstractCIMessageHandler

  /** The actor for which this handler processes messages */
  private IKfxParserActor m_oActor;

  /**
   * Sets the actor for which this handler processes messages.
   *
   * @param actor the IKfxParserActor
   */
  public void setActor(IKfxParserActor actor) {
    m_oActor = actor;
  } // setActor
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
/**
 * Returns the actor for which this handler processes messages.
 *
 * @return IKfxParserActor the actor
 */
public IKfxParserActor getActor() {
  return m_oActor;
} // getActor

public abstract CAPerformative getPerformative();

public abstract String getAction();

public abstract void process(ICommunicativeAct message);

} // AbstractCIMessageHandler
```

5.3.4 Spring Configuration

To associate message handler with the factory, and the factory with the actor we need to define and configure the factory and message handler as Spring beans and configure the appropriate properties to make the associations. The message handler is injected with its dependencies, the parser actor and the parser. The factory is injected with a set of message handlers with the set consisting of only the parseMessageHandler. The actor finally is injected with the kfxParserActorMessageHandlerFactory. Note that the actor injected into the message handler is actually the proxy, not the actual actor. As discussed earlier injecting an abstract bean (the actual actor) is not allowed in Spring.

```
<bean id="parseMessageHandler"
class="gov.nasa.arc.ci.spring.actor.parser.communication.ParseMessageHandler">
  <property name="actor" ref="kfxParserActor" />
  <property name="parser" ref="kfxParser" />
</bean>

<bean id="kfxParserActorMessageHandlerFactory"
class="gov.nasa.arc.ci.spring.common.communication.CIMessageHandlerFactory">
  <property name="handlers">
    <set>
      <ref bean="parseMessageHandler" />
    </set>
  </property>
</bean>

<bean id="kfxParserActorImpl"
  class="gov.nasa.arc.ci.spring.actor.parser.KfxParserActor"
  abstract="true">

  <property name="actorProxy" ref="kfxParserActor" />
  <property name="messageHandlerFactory"
       ref="kfxParserActorMessageHandlerFactory" />
  <property name="parser" ref="kfxParser" />
</bean>
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

That completes all of the coding and configuration of the actors and application logic. The last thing to do is to package the application so that we can run both actor hosting environments and their actors.

6 Packaging

To make the example run you will need to build the code and setup the necessary scripts and configuration files to run the example.

Normally this example is to be run on two different systems, but it can easily be configured to run both environments on a single machine. In this section we will configure the setup to run both on a single system. We separate out the two environments by client type, one client type running the user interface is referred to as the MAS client, the client type running the KFX parser is referred to as the OCA client. The scripts and configuration files will be separated by these client types.

6.1 Prerequisites

In order for you to run the application you must install the CI Runtime version 1.10. Contact the author for instructions on how to obtain this CI Runtime. In our instance we installed the CI Runtime in the directory: C:\Brahms\CI

The source package for the CI Spring Actors applications is located in CVS and includes all necessary dependency libraries required to run the applications. It includes an Ant build script to build the code.

The following libraries were used:

- Log4J 1.2.15
- JacORB 2.3.0
- Apache Commons Pool 1.5.4
- Apache Commons DBCP 1.2.2
- Hibernate 3.3.2 w/ its dependencies
- Hibernate Tools
- Spring Framework 2.5.6 w/ its dependencies
- MySQL Connector for Java 5.1.6
- CI Spring Interface 1.0.6
- CI Runtime 1.10

MySQL 5.1 was used as the DBMS.

6.2 File Organization

We installed the application in the following directory:

C:\Brahms\CISpring

In this directory we have the following sub folders and files:

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

bin\mas\ (startup script for MAS client)
CIHostingEnvironment.bat
bin\oca\ (startup script for OCA client)
CIHostingEnvironment.bat
config\mas (MAS Hosting Environment configuration files)
HostingEnvironment.cihx
HostingEnvironmentACE.cfg
HostingEnvironmentLogger.cfg
config\mas\heasp (MAS CI ASP and Services configuration files)
ASP.ciaspx
DataDistributionService.citx
DirectoryService.cidx
TranslationService.cilx
TransportService.cicx
config\oca (OCA Hosting Environment configuration files)
HostingEnvironment.cihx
HostingEnvironmentACE.cfg
HostingEnvironmentLogger.cfg
config\oca\heasp (OCA CI ASP and Services configuration files)
ASP.ciaspx
DataDistributionService.citx
DirectoryService.cidx
TranslationService.cilx
TransportService.cicx
deploy\ (actors jar file)
cispringactors.jar
deploy\mas (MAS Actor and Spring configuration files)
GuiActor.cfg
GuiActor.ciax
hibernate.properties
jdbc.properties
mas-applicationContext.xml
masBeanRefContext.xml
deploy\mas\META-INF (MAS AOP configuration file)
aop.xml
deploy\oca (OCA Actor and Spring configuration files)
KfxActor.cfg
KfxActor.ciax
hibernate.properties
jdbc.properties
mas-applicationContext.xml

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

masBeanRefContext.xml

deploy\oca\META-INF (OCA AOP configuration file)

aop.xml

lib\ (all dependent jar files, spring libraries, hibernate libraries, etc)

antlr-2.7.6.jar
aspectjrt.jar
aspectjweaver.jar
cispringinterface.jar
commons-collections-3.1.jar
commons-dbcp-1.2.2.jar
commons-logging.jar
commons-pool-1.5.4.jar
dom4j-1.6.1.jar
freemarker.jar
hibernate3.jar
hibernate-tools.jar
javassist-3.9.0.GA.jar
jaxen-1.1-beta-7.jar
jta-1.1.jar
mysql-connector-java-5.1.6.bin.jar
slf4j-api-1.5.0.jar
slf4j-log4j12-1.5.0
spring.jar
spring-agent.jar
spring-aspects.jar

lib\resources (spring resources for local loading, instead of the internet)

spring.ftl
spring.tld
spring.vm
spring-aop-2.0.xsd
spring-aop-2.5.xsd
spring-beans.dtd
spring-beans-2.0.dtd
spring-beans-2.0.xsd
spring-beans-2.5.xsd
spring-context-2.5.xsd
spring-form.tld
spring-jee-2.0.xsd
spring-jee-2.5.xsd
spring-jms-2.5.xsd
spring-lang-2.0.xsd

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

spring-lang-2.5.xsd
spring-tool-2.0.xsd
spring-tool-2.5.xsd
spring-tx-2.0.xsd
spring-tx-2.5.xsd
spring-util-2.0.xsd
spring-util-2.5.xsd
logs (directory to write the log files to)

6.3 Configuring the Actor Hosting Environments

For each Actor Hosting Environment a number of files need to be included in the distribution to properly configure that Actor Hosting Environment. The configuration includes the Actor Hosting Environment configuration file itself (cihx) and its logger configuration files and the configuration of the CI Actor Service Provider (ASP) and its services as used by the actor hosting environment.

The Actor Hosting Environment requires for elements to be configured:

- Names; simple name, qualified name, and directory names
- Location of the ASP descriptor
- Descriptors of the actors to load in the actor hosting environment
- The deployment directory in which actor descriptors can be found

The configuration file for the MAS Hosting Environment is listed below. OCA's hosting environment is almost identical except for the different names, actors to load and deployment directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<HOSTING_ENVIRONMENT>
  <!-- The credentials of the hosting environment itself -->
  <CREDENTIALS>
    <!-- The simple name for the hosting environment, used for display or
         logging purposes -->
    <SIMPLENAME>MASHostingEnvironment</SIMPLENAME>

    <!-- The globally unique fully qualified name of the hosting
          environment -->
<QUALIFIEDNAME>gov.nasa.arc.ci.spring.he.MASHostingEnvironment</QUALIFIEDNAME>

    <!-- The directory name(s) with which the hosting environment is
         registered in the directory service. -->
    <DIRECTORYNAMES>
<DIRECTORYNAME>gov.nasa.arc.ci.spring.he.MASHostingEnvironment</DIRECTORYNAME>
<DIRECTORYNAME>ci/spring/hostingenvironment/MASHostingEnvironment</DIRECTORYNAM
E>
    </DIRECTORYNAMES>

    <!-- Other credentials properties -->
    <PROPERTIES>
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    </PROPERTIES>
  </CREDENTIALS>

  <!-- The hosting environment's advertisement used to publish its
       capabilities -->
  <ADVERTISEMENT>
    <!-- The list of capabilities published by the hosting environment -->
    <CAPABILITIES>
      <CAPABILITY>
        <NAME>Hosting</NAME>
        <KEYWORDS>
          <KEYWORD>Hosting</KEYWORD>
          <KEYWORD>Environment</KEYWORD>
        </KEYWORDS>
        <DESCRIPTION>
          Provides an environment to host one or more actors.
        </DESCRIPTION>
      </CAPABILITY>
    </CAPABILITIES>

    <!-- Other advertisement properties -->
    <PROPERTIES>
    </PROPERTIES>
  </ADVERTISEMENT>

  <!-- The actor service provider used to configure the CI services
       provided to the hosting environment and the actors in the
       hosting environment. -->
  <ASP>
    <DESCRIPTOR>heasp/ASP.ciaspx</DESCRIPTOR>
  </ASP>

  <!-- The list of actors that need to be loaded by the hosting environment.
       This allows for a set of actors to be loaded when the hosting
       environment is created. It is possible to load actors at a
       later time using the hosting environment's management service. -->
  <ACTORS>
    <ACTOR>
      <DESCRIPTOR>GuiActor.ciax</DESCRIPTOR>
    </ACTOR>
  </ACTORS>

  <!-- General configuration properties -->
  <PROPERTIES>
    <!-- The hosting environment's deployment directory in which to search
         for actor descriptors.
    -->
    <PROPERTY name="ci.ahe.deploydir">C:/Brahms/CISpring/deploy/mas</PROPERTY>
  </PROPERTIES>
</HOSTING_ENVIRONMENT>
```

The HostingEnvironmentACE.cfg file is intended only for C++ based actors hosted in the Hosting Environment and configures the CORBA ORB and logging service.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

The HostingEnvironmentLogger.cfg file is used to configure the Log4J logging. See Log4J documentation for more information about how to configure Log4J logging [5].

The ASP descriptor specifies which CI services need to be loaded and configured and whether any external services or frameworks need to be integrated. In our case we need to add the `CISpringInterface` ASP Hook for Spring integration and define the related properties to load the appropriate application contexts by setting the `ci.spring.selector` and `ci.spring.contexts` properties. The following is the ASP descriptor for MAS. For OCA the only difference is the value for the `ci.spring.selector` property, it is in that case `classpath*:ocaBeanRefContext.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>

<ASP>
  <!-- The transport service to be loaded, configured and used by the
       actor service provider. Specifies the name of the file with the
       transport service configuration. This service configures the
       shared endpoints shared with all actors. -->
  <TRANSPORT_SERVICE>
    <DESCRIPTOR>TransportService.cicx</DESCRIPTOR>
  </TRANSPORT_SERVICE>

  <!-- The directory service to be loaded, configured and used as
       part of the virtual network. Specifies the name of the file
       with the directory service configuration.
       Note: If the directory service is declared also the data distribution
       service must be declared. -->
  <DIRECTORY_SERVICE>
    <DESCRIPTOR>DirectoryService.cidx</DESCRIPTOR>
  </DIRECTORY_SERVICE>

  <!-- The data distribution service to be loaded, configured and used
       as part of the virtual network. Specifies the name of the file
       with the data distribution service configuration. -->
  <DATA_DISTRIBUTION_SERVICE>
    <DESCRIPTOR>DataDistributionService.citx</DESCRIPTOR>
  </DATA_DISTRIBUTION_SERVICE>

  <!-- The translation service to be loaded, configured and used by the
       actor service provider. Specifies the name of the file with the
       translation service configuration. -->
  <TRANSLATION_SERVICE>
    <DESCRIPTOR>TranslationService.cilx</DESCRIPTOR>
  </TRANSLATION_SERVICE>

  <!-- (Optional) The Actor Service Provider hooks to be woven
       into the actor service provider to allow for external services
       or frameworks to be integrated with the actor service provider.
       For each hook the qualified Java class name must be specified.
       This java class must implement the gov.nasa.ci.api.asp.IASPHook
       interface.
  -->
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
    <ASP_HOOKS>
      <ASP_HOOK>gov.nasa.ci.spring.CISpringInterface</ASP_HOOK>
    </ASP_HOOKS>

    <!-- General configuration properties -->
    <PROPERTIES>
      <!-- For each property the name and value -->

      <!-- The Spring selector specifies the location of the resource(s)
           which will be read and combined to form the definition of the
           BeanFactoryLocator instance. Default: classpath*:beanRefContext.xml
      -->
      <PROPERTY
  name="ci.spring.selector">classpath*:masBeanRefContext.xml</PROPERTY>

      <!-- The Spring contexts defines a comma delimited list of contexts
          to be initialized.
      -->
      <PROPERTY name="ci.spring.contexts">ocams-context</PROPERTY>


      <!-- The actor service provider's configuration directory in which
           to search for the service descriptors.
      -->
      <PROPERTY name="ci.asp.dir.config">.</PROPERTY>
    </PROPERTIES>
  </ASP>
```

For details about configuring the services please see the CI Tutorial for Java [6]. In those configuration the main point is to properly configure the endpoints and to make sure that the names for the services are properly configured and unique.

6.4 Actor Hosting Environment Startup Script

The startup script for the Actor Hosting Environment specifies which configuration directories need to be used to ensure that the proper Actor Hosting Environment configuration file is used to start and configure the Actor Hosting Environment. The following script used as the basis for this application is based on the original hosting environment script included with the CI Runtime.

```
@ECHO OFF
SET CI_ROOT=@CI_ROOT@
SET CI_SPRING_ROOT=@CI_SPRING_ROOT@
SET CLIENT_TYPE=@CLIENT_TYPE@
SET PATH=%ACE_ROOT%\lib;%CI_ROOT%\lib;%CI_ROOT%\lib\nss;%CI_ROOT%\deploy;%PATH%

REM Java Setup
SET JAVA_MEMORY=-Xincgc -Xmx768m -Xss1024k -Xms32m
SET BOOTCP=-Xbootclasspath/p:"%CI_ROOT%\lib\jacorb.jar;%CI_ROOT%\lib\logkit-
1.2.jar;%CI_ROOT%\lib\avalon-framework-4.1.5.jar"
SET CP=-cp
".;%CI_SPRING_ROOT%\lib\resources;%CI_SPRING_ROOT%\config\%CLIENT_TYPE%;%CI_SPR
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

```
ING_ROOT%\deploy\%CLIENT_TYPE%;%CI_SPRING_ROOT%\deploy;%CI_ROOT%\config;%CI_ROO
T%\config\security;%CI_ROOT%\deploy"
SET EXT_JAR_DIRS=-
Djava.ext.dirs="%CI_ROOT%\jre\lib\ext;%CI_ROOT%\lib;%CI_ROOT%\lib\jidesoft;%CI_
ROOT%\lib\apache;%CI_ROOT%\lib\nss;%CI_SPRING_ROOT%\lib;%CI_SPRING_ROOT%\deploy
\%CLIENT_TYPE%;%CI_SPRING_ROOT%\deploy;%CI_ROOT%\deploy"
SET JAVA_LIB_PATH=-
Djava.library.path="%ACE_ROOT%\lib;%CI_ROOT%\lib;%CI_ROOT%\lib\nss"
SET ORB=-Dorg.omg.CORBA.ORBClass=org.jacorb.orb.ORB -
Dorg.omg.CORBA.ORBSingletonClass=org.jacorb.orb.ORBSingleton
SET ASPECTJ=-javaagent:%CI_SPRING_ROOT%\lib\aspectjweaver.jar
SET JAVA_PROPERTIES=%JAVA_MEMORY% %BOOTCP% %CP% %EXT_JAR_DIRS% %JAVA_LIB_PATH%
%ORB% %ASPECTJ%

title @CLIENT_TYPE@ Client Hosting Environment
"%CI_ROOT%\jre\bin\java.exe" %JAVA_PROPERTIES%
gov.nasa.ci.ahe.HostingEnvironment HostingEnvironment
```

Three variables need to be substituted with the appropriate values:

- @CI_ROOT@; with the value for the CI Runtime installation directory (C:\Brahms\CI in our case).
- @CI_SPRING_ROOT@; with the value for the CI Spring Actors installation directory (C:\Brahms\CISpring in our case).
- @CLIENT_TYPE@; one of *mas* or *oca*, this ensures that the proper configuration and deployment directories are placed in the classpath.

By passing the value 'HostingEnvironment' as an argument to the Java application we indicate to the Hosting Environment that it needs to search for a file named HostingEnvironment.cihx for the configuration of the hosting environment, and the files HostingEnvironmentACE.cfg and HostingEnvironmentLogger.cfg for the configuration of the C++ interface and the Log4J logger. So the value passed is the base filename for the configuration files for that hosting environment causing the hosting environment to look for <name>.cihx, <name>ACE.cfg and <name>Logger.cfg.

6.5 Spring Selector Configuration File

In this document we focused on setting up the main application context configuration file for Spring in which all beans are configured. As we saw in the ASP descriptor's `ci.spring.selector` property we also need to create a beanRefContext file that specifies which application context files to load. For the MAS client this is the masBeanRefContext.xml, for the OCA client the ocaBeanRefContext.xml. The files are identical except for the filename of the application context configuration file that needs to be loaded, for MAS this is mas-applicationContext.xml for OCA this is oca-applicationContext.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<!-- load a hierarchy of contexts, although there is just one here -->
<beans>
```

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

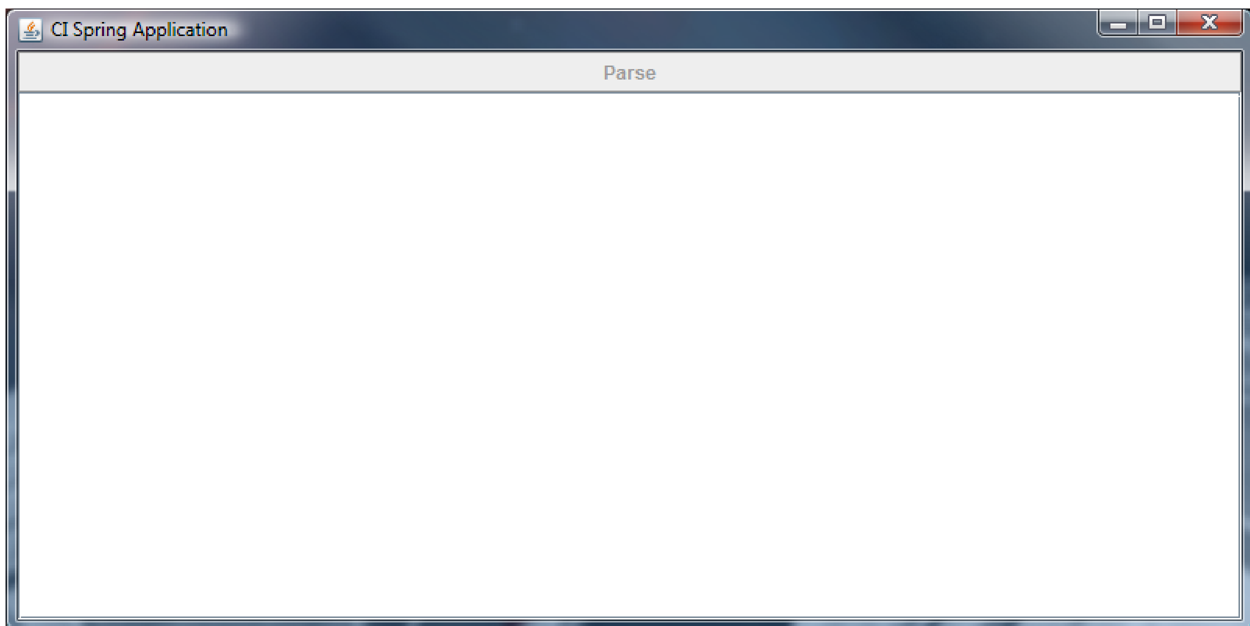
```
  <bean id="ocams-context"
class="org.springframework.context.support.ClassPathXmlApplicationContext">
    <constructor-arg>
      <list>
        <value>/mas-applicationContext.xml</value>
      </list>
    </constructor-arg>
  </bean>
</beans>
```

The bean identifier (ocams-context) is the name of the context that is referred to in the ASP descriptor's `ci.spring.contexts` property as the context to be loaded when the ASP's Spring Interface hook is initialized to integrate and configure Spring. This will trigger for Spring to process the application context file. See [1] for more information about this file and the application context file.

7 Running the Application

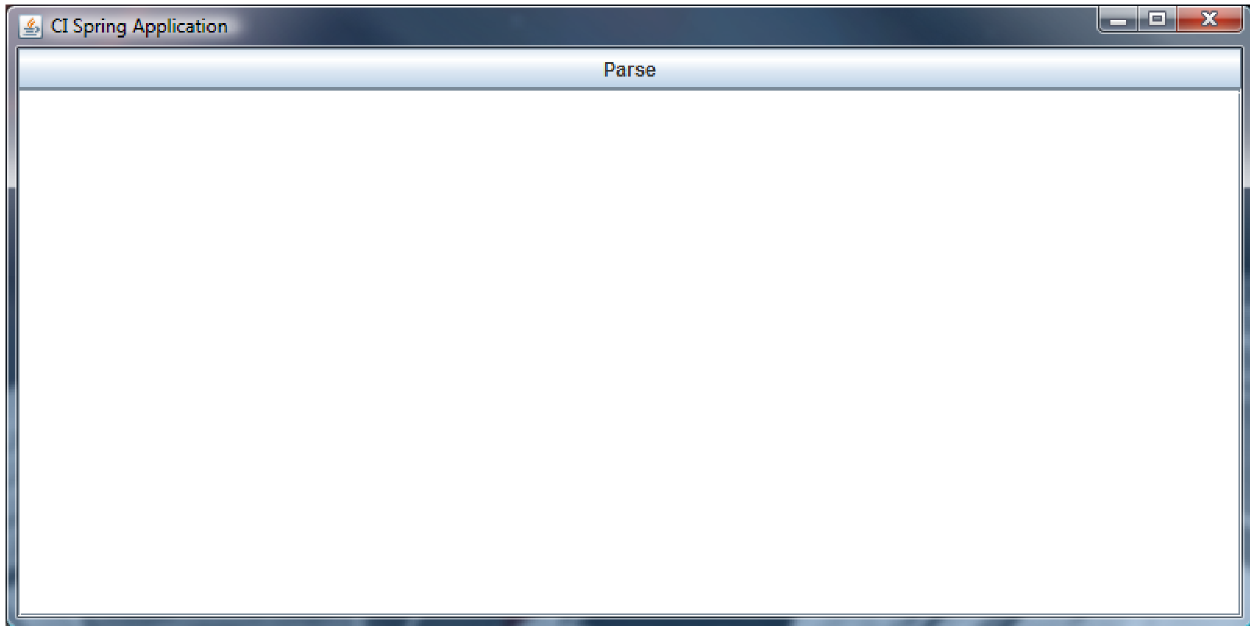
To run the application start the CIHostingEnvironment script in the bin\mas and bin\oca directories, starting the script in the bin\mas directory first. Remember that we configured the Spring application context for MAS to create the table structures in the database on startup.

When starting the MAS CI Hosting Environment you will see the user interface being displayed with a Parse button that is grayed out. This button is only enabled when the KFX Parser Actor is found.

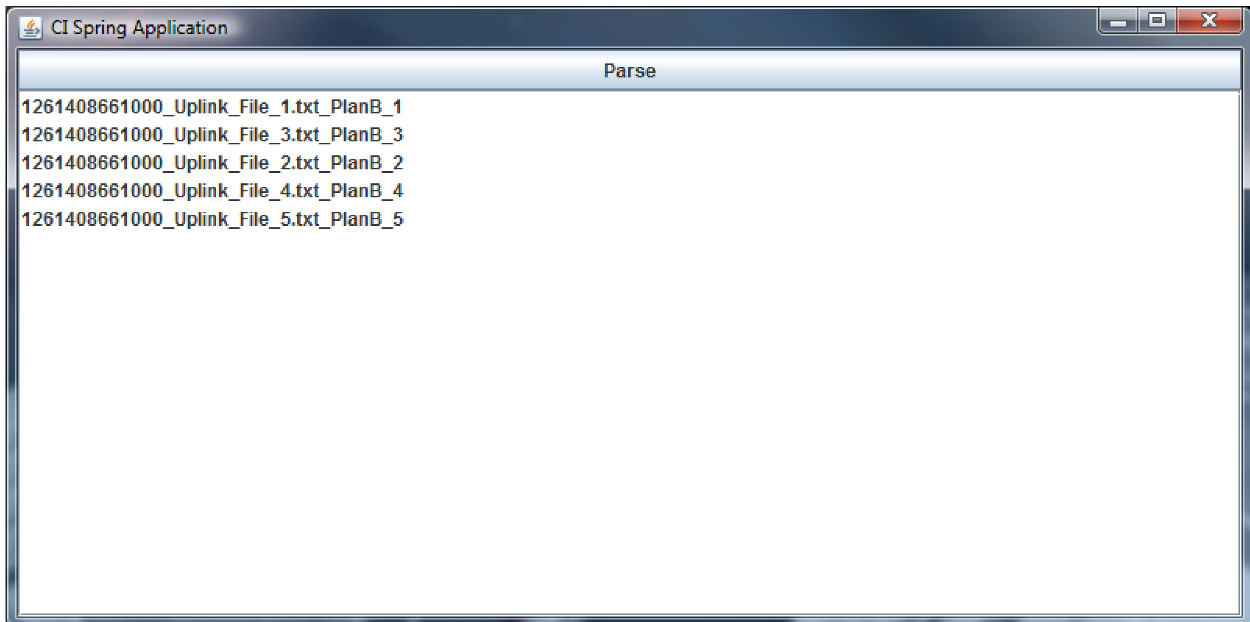


Now when starting the OCA CI Hosting Environment you will see the Parse button become enabled once that hosting environment is fully started and the KFX Parser Actor registered itself.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.



After pressing the Parse button you will see the list of keys being displayed for the parsed KFX files as they are received from the KFX Parser Actor and retrieved from the data store.



8 Special Notes

8.1 AOP.xml

To improve the performance of the application at startup it is possible to indicate to Spring and AspectJ to which packages to limit the search for Configurable beans. This is done by including the file aop.xml in a META-INF folder in the classpath. This file lists the packages to include and/or exclude in the search.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.

For the MAS/GUI application this file's content is:

```
<aspectj>
  <weaver>
    <include within="gov.nasa.arc.ci.spring.actor.gui.*" />
    <exclude within="gov.nasa.arc.ci.spring.actor.gui.communication.*" />
  </weaver>
</aspectj>
```

For the OCA/KFX Parser application this file's content is:

```
<aspectj>
  <weaver>
    <include within="gov.nasa.arc.ci.spring.actor.parser.*" />
    <exclude within="gov.nasa.arc.ci.spring.actor.parser.communication.*" />
  </weaver>
</aspectj>
```

9 References

[1] Integrating Spring with the Collaborative Infrastructure, Version 1.0 12/16/2009, Ron van Hoof

[2] Spring in Action, Second Edition, Craig Walls

[3] Spring Reference Documentation: http://static.springsource.org/spring/docs/2.5.x/reference/

[4] Hibernate Reference Documentation: http://docs.jboss.org/hibernate/stable/core/reference/en/html/

[5] Log4J Documentation: http://logging.apache.org/log4j/1.2/manual.html

[6] CI Tutorial Java, Version 1.1 4/22/2008, Ron van Hoof

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2009 NASA Ames Research Center. All Rights Reserved.