Java Design Pattern Framework (Jt)

Version 7.6

Table of Contents

1. Overview

This document describes Jt, a Design Pattern Framework for the rapid implementation of Java and Android applications. Jt has been utilized in several large mission critical systems. The framework addresses the following goals:

A) The design pattern framework implements and/or facilitates the implementation of well-known design patterns like Gang of Four design patterns (GoF) and J2EE Design patterns. The framework itself is conceived and implemented, from the ground up, based on design patterns. The framework also facilitates and accelerates the implementation of applications based on design patterns.

B) The framework architecture is based on a messaging design pattern: framework components are able to interchange information and perform computations by sending, receiving and processing messages. A messaging API provides simplicity, strong encapsulation and loose coupling; framework components can be interchangeably plugged into complex framework applications using a “lego/messaging” architecture. Framework messages can be processed synchronously or asynchronously. The framework takes full advantage of the power and simplicity of the messaging design pattern/API.

C) The framework lego/messaging architecture provides transparent access to remote components: remote framework objects are treated as local objects. Design patterns implemented by the framework (adapters, remote proxies and facades) make this possible by hiding the complexities associated with remote APIs.

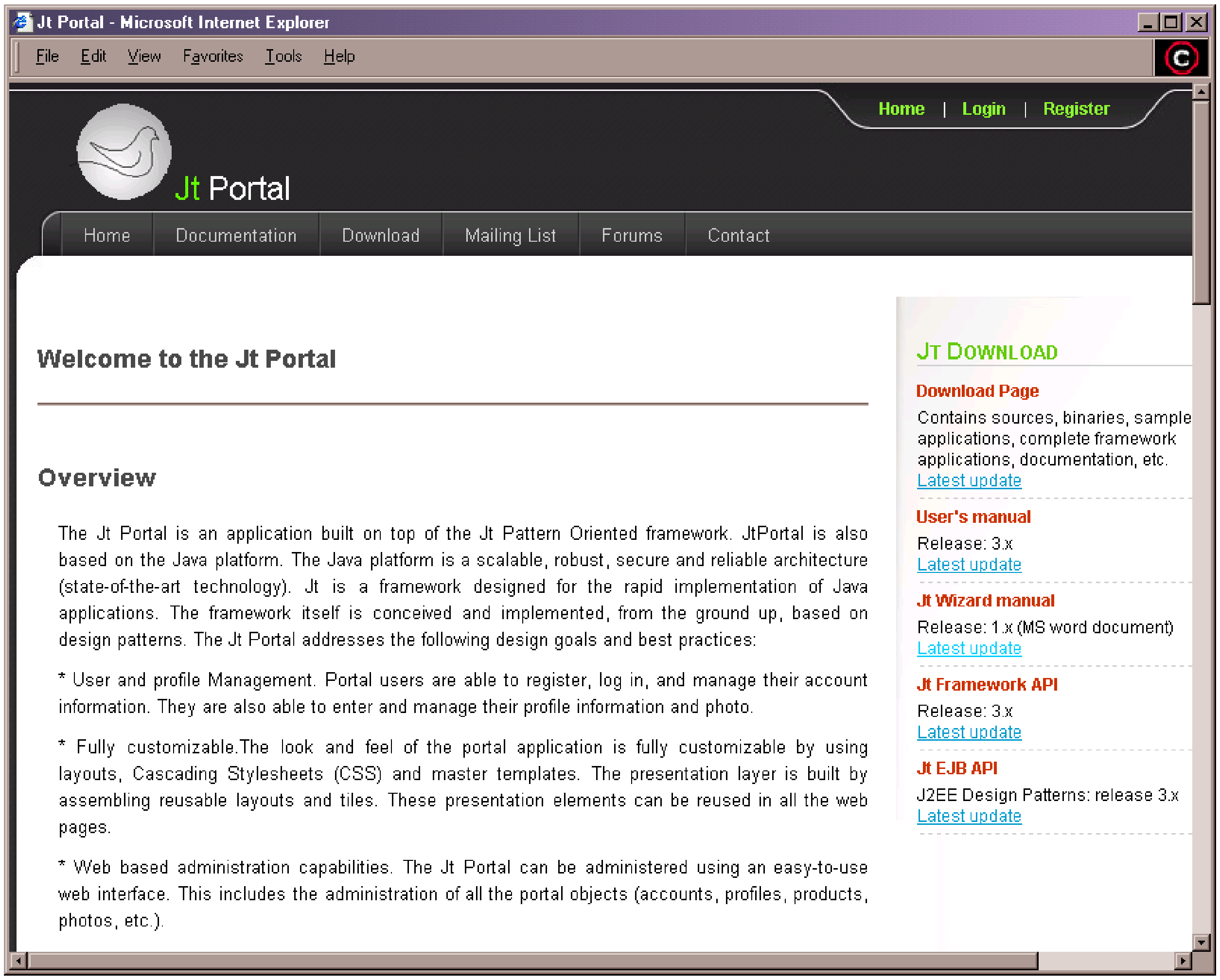
D) The framework provides transparent integration with other technologies via framework adapters, proxies and the implementation of related design patterns. These technologies include Android, BPEL, BPM, Data Access Object implementations (DAO), Model View Controller implementations (MVC), EJBs, JSP, AJAX, ESB, JMS, XML, REST and Web Services.

E) The framework is designed to be lightweight and fast (low overhead/small footprint). The main Jt functionality is able to run on smartphones under Android.

F) The framework messaging/lego architecture should improve and simplify design/development efforts. There is a tight correspondence between UML design diagrams and the framework messaging based applications and components needed for the implementation. The framework provides wizards and automated capabilities for generating framework applications. Framework components can be easily added to BPEL/BPM process diagrams. In future versions of the framework, it should be possible for application modules to be generated directly from the UML design diagrams. This goal is still work in progress.

G) The framework messaging architecture facilitates testing and debugging efforts. The framework provides capabilities for testing components as independent units by sending messages to the component and verifying the expected reply messages.

The framework distribution includes several reference applications built on top of the Jt Pattern Oriented framework. These are complete production quality applications. The JtPortal application implements portal and electronic commerce capabilities: user profile, mailing list, portal administration, shopping cart, forums, chat, etc. The Jt Wizard is an application that provides automated capabilities for generating framework applications. The Jt Wizard is able to automatically generate application modules based on several design patterns including Jt Messaging, DAO, MVC and Gang of Four. The JtRftp application allows users to transfer files of any size reliably by taking advantage of the framework Enterprise Service Bus (ESB) capabilities and components. The use of JMS and transactions make the file transfer reliable.


Jt Portal - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Home | Login | Register
Jt Portal
Home Documentation Download Mailing List Forums Contact

Welcome to the Jt Portal

Overview

The Jt Portal is an application built on top of the Jt Pattern Oriented framework. JtPortal is also based on the Java platform. The Java platform is a scalable, robust, secure and reliable architecture (state-of-the-art technology). Jt is a framework designed for the rapid implementation of Java applications. The framework itself is conceived and implemented, from the ground up, based on design patterns. The Jt Portal addresses the following design goals and best practices:

* User and profile Management. Portal users are able to register, log in, and manage their account information. They are also able to enter and manage their profile information and photo.

* Fully customizable.The look and feel of the portal application is fully customizable by using layouts, Cascading Stylesheets (CSS) and master templates. The presentation layer is built by assembling reusable layouts and tiles. These presentation elements can be reused in all the web pages.

* Web based administration capabilities. The Jt Portal can be administered using an easy-to-use web interface. This includes the administration of all the portal objects (accounts, profiles, products, photos, etc.).

JT DOWNLOAD

Download Page
Contains sources, binaries, sample applications, complete framework applications, documentation, etc.
Latest update

User's manual
Release: 3.x
Latest update

Jt Wizard manual
Release: 1.x (MS word document)
Latest update

Jt Framework API
Release: 3.x
Latest update

Jt EJB API
J2EE Design Patterns: release 3.x
Latest update

2. Messaging Design Pattern (MDP)

This design pattern was published in the 17th conference on Pattern Languages of Programs (PLoP 2010).

Intent: The messaging design pattern allows the interchange of information (i.e. messages) between components and applications. It improves decoupling, encapsulation and reusability by separating component communication from component functionality.

Messaging is ubiquitous. On the other hand, conventional software methodologies and models tend to overlook it. MDP addresses this problem by proposing and relying on a messaging model. While designing and manufacturing software, we need to think not only in terms of software components but also in terms of the messaging being exchanged between these entities. Components are one essential part of the object oriented methodologies. The interchange of information (i.e. messaging) between components is also an important part of a complete model. This helps justify the need for MDP and a model based on a messaging approach. It also illustrates the gap between conventional models and the reality that they seek to represent. MDP addresses this gap.

Applicability: This design pattern can be applied to solve a great variety of problems in many diverse scenarios. A messaging paradigm is widely used in the real world. Messages are interchanged all around us. Entities are constantly sending, receiving and processing messages. Human beings for instance: when we watch TV, when we talk to a friend, talk over the phone, or send an email message. Right now, you are reading this written message which is possible because of messaging. Since computer applications seek to model the real world, it is only natural to design and write applications using a messaging approach. We can argue that messaging provides a better and more accurate representation (i.e. model) of the real world.

Motivations: Object oriented applications consist of a collection of components that need to communicate with each other. As proposed hereafter, this interaction can be accomplished via a messaging (MDP) model. This is consistent with the real world in which independent entities interface with each other via messaging. Each entity is a self-contained/independent unit. The mechanism of communication with other entities is via messaging. As a consequence, MDP minimizes coupling. MDP also improves encapsulation and reusability.

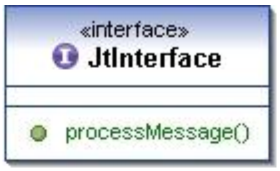
Conventional implementations, not based on a messaging model, are unable to provide this level of encapsulation, decoupling and reusability. MDP and the messaging model address these shortcomings while at the same time reducing complexity. Actually we can argue that messaging should be the principal mechanism of communication since components are independent entities. Therefore they should be modeled and treated as such, in order to accurately mimic reality. A later section will demonstrate how design patterns implemented using MDP are combined to implement access to distributed components. Conventional models not based on MDP present complexities and limitations. MDP addresses these problems.

Participants:

a) Message Sender: Component that sends the message.
b) Message Recipient (Receiver): Component that receives the input message and may produce a reply (output message) after processing it. The input message, general in nature, may contain any type of information. The component may be instructed to perform computations based on the input message.
c) Messenger: Intermediary that transfers the message from the sender to the recipient. The sender and the recipient don't need to be concerned about how the message is transferred (communication protocol, message format, encryption/security mechanism, etc.) and the transformations performed on the message along the way. This is the messenger's purpose and responsibility. Similar to the real world, it is often the case that the messenger is not required. The message can be sent directly to the message recipient. Several modes of communication are possible: synchronous, asynchronous and two-way messaging.
d) Message: any piece of information (i.e. data) that needs to be interchanged between sender and recipient. Two messages are usually involved: input message and output message (or reply message). The reply message is not required.

Structure:

The messaging design pattern is implemented using the messaging interface (JtInterface). This interface consists of a single method to process the input message and produce a reply message.

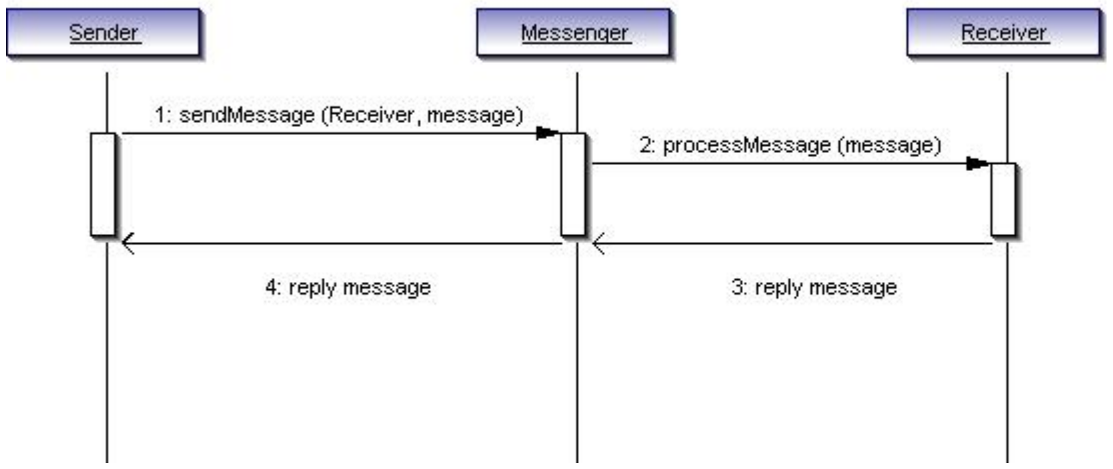


Messaging Interface

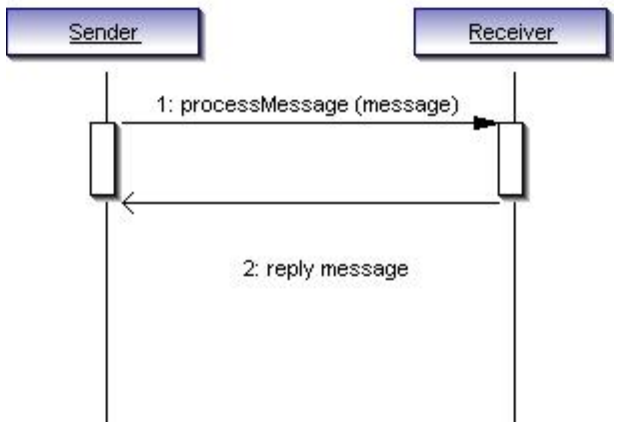
The MDP messaging interface is simple but powerful. The simplicity of this interface can be deceiving. One method with one parameter (message) is all that is needed. It acts as a universal messaging interface that applies to remote and local components. This interface handles any type of message (For instance, the Java Object class). It returns a reply (of any type). In Java, the messaging interface can be declared as follows:

public interface JtInterface { Object processMessage (Object message); }

The message receiver needs to implement this interface in order to receive and process incoming messages. Although Java is used here, MDP can be implemented using any comparable computer language. Languages that don't use interfaces can simply declare a processMessage() function or method in order to implement MDP.


Sender
Messenger
Receiver
1: sendMessage (Receiver, message)
2: processMessage (message)
4: reply message
3: reply message

Messaging Design Pattern (synchronous mode).



Messaging Design Pattern (synchronous mode without messenger involved)

Consequences:

Encapsulation. The messaging design pattern maximizes encapsulation. As mentioned earlier, each component is a self-contained/independent unit. The only mechanism of communication with other components and applications is via messaging.

Decoupling. MDP minimizes coupling. Again each component is a self-contained unit that can perform independently from the rest of the system.

Reusability. MDP improves reusability. This is similar to the building blocks in a “Lego” set. Very complex models can be built based on simple pieces that share a simple way of interconnecting them (i.e. common interface). The power of the approach is derived from the number of combinations in which these toy pieces can be assembled. Components that use MDP can be interchangeably plugged into complex applications. The components can be assembled in a limitless variety of configurations. The user of a component only needs to know the input/output messages that the component handles. Applications are also able to reuse components from other applications at the component level: a single component can be extracted from another application, provided that the messaging design pattern is being used.

QA/Testing process. MDP facilitates testing and debugging efforts. Components are tested as independent units by sending messages to the component and verifying the expected reply messages (black-box testing). In general, unit testing can be performed via a testing harness. No need to include testing code inside the component code which can be time consuming and lead to the unexpected introduction of software defects.

Design process. MDP improves and simplifies the design process. The bulk of the design work becomes defining the set of components needed to meet the system requirements and the input/output messages that each component needs to handle. There is a tight correspondence between UML design diagrams and the components needed for the implementation. Several UML diagrams are geared towards messaging (sequence and collaboration) although their implementation doesn't rely on it. The UML model and the implementation are disconnected when MDP is not used. Since all components share the same messaging interface, they can also be readily added to BPM/BPEL diagrams. As mentioned earlier, this is similar to building blocks that can be reused and connected in many different ways.

Development process. Since each component that relies on messaging is self-contained, a large team of people can cooperate in the development effort without stepping on each other's work/code. In the ideal situation, responsibility for one component/package can be given to an individual. The rest of the team only needs to know the input/output messages that someone else's component is designed to handle. In general, there is no need to change someone else's code. The need for creating, maintaining and merging several versions of the code is also minimized or eliminated. Testing/QA engineers can perform their testing independently via a testing harness. As a general rule, there is no need to add testing code to the component itself.

Logging and Debugging. Since all the components use the same messaging interface, messages can be logged automatically. This minimizes the need for print/logging statements inside the code which can be time consuming and error-prone. By taking a look at the messages being interchanged and automatically logged, the user is usually able to quickly track down the message/component that is causing the problem (with minimum or no extra effort).

Security. Well-known encryption and authentication mechanisms fit in well with the messaging design pattern. Strong security can be provided by the framework that implements MDP [3]. This is done by encrypting and authenticating the messages being interchanged. The sender and the recipient don't need to be too concerned with how secure messaging is implemented. This provides strong security while at the same time simplifying the implementation of security. If required, custom security mechanisms can also be incorporated: sender and receiver need to agree on and implement the message encryption/authentication mechanism to be used.

Multithreading and asynchronous messaging. MDP is able to handle the complexities associated with multithreading and asynchronous messaging. Components that implement MDP are able to execute in a separate/independent thread. This is a natural representation of the real world: each component (entity) is a self-contained unit and executes independently for the rest of the system. Messages can be processed asynchronously using the component's own independent thread. This capability is usually implemented in the context of a component framework [3]. The component doesn't need to add separate logic to handle multithreading which is time consuming, complex and prone to error.

Speed of development and cost. Because of all the reasons outlined above, the messaging design pattern is able to substantially improve the speed of development and reduce cost.

Quality and software maintenance. Quality and software maintenance efforts are also improved as a result of the all of the above.

Messaging paradigm and learning curve. In order to take full advantage of this design pattern, people need to think in terms of a messaging paradigm when they model, design and build software applications: independent entities (i.e. components) interchanging messages among each other. This may require learning time and training. Although a messaging approach is natural, intuitive, and consistent with the real world, traditional approaches are based on method/procedure invocation (both local and remote).

Overhead. Transferring messages between components introduces a small overhead when compared with traditional method/procedure invocation. This is especially true when a messenger is used. As computers become faster and faster this becomes a non-issue. Also, the benefits of messaging outweigh this small performance cost. Notice that the messenger component is part of many real world problems and applications in which an intermediary is necessary for message interchange.

Disciplined approach. MDP encourages a disciplined approach that may have a small impact on the initial development time of a component. Messaging should be the only channel of communication between components. External class methods may still be invoked using the traditional approach. On the other hand, this should be used sparingly in order to minimize coupling and maximize encapsulation. An ideal component is a self-contained unit that interacts with the other components only via messaging. The additional development time is again outweighed by the benefits introduced by messaging. Moreover individual components based on messaging can be purchased or extracted from other applications.

Known uses

Design patterns. MDP has been used to implement and/or facilitate the implementation of other well-known design patterns like Gang of Four design patterns (GoF), DAO, J2EE Design patterns, etc. MDP also provides a more natural, streamlined and straightforward implementation of other design patterns. This topic is covered in detail later in this paper.

Distributed component and messaging model. MDP is particularly well suited for the implementation of a complete distributed component model. It is able to provide transparent access to remote components regardless of the protocol, technology and communication mechanism being used: remote objects are treated as local objects. Messages can be transferred via web services, REST, EJBs, HTTP, sockets, SSL or any comparable communication interface. Design patterns implemented using messaging (adapters, remote proxies and facades) make this possible by hiding the complexities associated with remote APIs. MDP solves a whole set of problems dealing with remote application interfaces and remote access to distributed components. Because of MDP, sender and recipient don’t need to be concerned with how messages are transferred.

Component based frameworks and design/BPM/BPEL tools. MDP can be utilized to implement component based frameworks [3]: components can be interchangeably plugged into complex framework applications using the “Lego” architecture previously described. These components can also be readily incorporated into UML/BPM/BPEL diagrams in order to design and implement complex applications. Notice that for components to be used interchangeably, they need to share the same interface. MDP provides this common messaging interface.

Secure Web Services. MDP has been utilized to implement secure web services. This includes RESTful web services. A Web service is just another mechanism of communication between heterogeneous applications. Notice that the messaging design pattern doesn’t place any restrictions on the message sender and recipient. These components can be running on multiple computers and operating systems. They can also be implemented using multiple computer languages and technologies.

Enterprise Service Bus (ESB) components and applications. Messaging has been used to implement ESB components and applications. Once all the building blocks are present (remote proxies, adapters, facades, etc), they can be assembled to create a new application in a fraction of the time required by traditional methods.

Secure and Multithreaded applications. MDP provides the building blocks required to assemble secure and multithreaded applications in a fraction of the time required by traditional methods. These building blocks are usually provided within the context of a messaging framework.

Fault-tolerant applications. MDP behaves like a state machine. Therefore it can be extended to provide fault-tolerant capabilities in a very natural fashion by replicating components and coordinating their interaction via consensus algorithms. Adding fault-tolerant characteristics to a program that uses a traditional approach is, in general, a difficult undertaking.

3. Jt Messaging API

3.1. Jt Messaging Interface (JtInterface)

The framework architecture is based on a messaging design pattern: framework components are able to interchange information and perform computations by sending, receiving and processing messages. Frameworks messages, general in nature, may contain any type of information. Framework components are instructed to perform computations based on these messages. After processing each message, a reply message is returned. The messaging architecture provides simplicity, stronger encapsulation and looser coupling. Notice that by using this messaging pattern, we don't need to know the methods implemented by a particular object or what methods and parameters are needed in order to perform a computation. We just need to know what messages the component is able to understand/process and what type of reply message should be expected. This is consistent with a black box type of behavior. Only the characteristics of the input and output messages are known.

The framework lego/messaging architecture provides transparent access to remote components: remote framework objects are treated as local objects. Design patterns implemented by the framework (adapters, remote proxies and facades) make this possible. We don't need to be concerned as to how the messages are transferred. Messages can be transferred via Web services, Ajax, JMS, EJBs, RMI, HTTP, sockets, SSL or any other API.

The messaging design pattern is implemented using the Jt messaging interface (JtInterface). This interface consists of a single method:

```java
public interface JtInterface  {
/**
  * Jt messaging interface used for the implementation
  * of the messaging design pattern.
  * Process a message and return a reply.
  */
  Object processMessage (Object message);
}
```

Advanced object oriented technologies provide features like Java generics which allow the types to be parameterized:

```java
 public interface MDPInterface<Type,Type1> {Type1 processMessage (Type msg);}
```

The JtInterface is simple but powerful. The simplicity of this interface can be deceiving. One method is all that is needed (dry). The framework architecture and implementation of its design patterns is based on JtInterface. It acts as a universal messaging interface that applies to remote and local framework components. This interface handles any type of message (Object class). External components can be integrated with the Jt framework by implementing this single interface which is just a simple matter of adding processMessage() to the external component.

3.2. Defining framework components

Framework components can be any Java class (pojo, Java bean, etc). There are two ways in which you can define new classes in the Jt framework: a) Extend JtObject or one of its subclasses. b) Implement the interface JtInterface.

JtObject is the top-level class of the Jt Framework hierarchy. JtObject implements JtInterface.

A very simple class can be used to illustrate the definition of new framework classes. We'll build our Hello World class step by step. The initial implementation of a HelloWorld class would look like the following:

```
import Jt.*;
public class HelloWorldMessage extends JtObject {
    public static final String JtCLASS_NAME = HelloWorldMessage.class.getName();

    private String greetingMessage = "Hello World ... Welcome to Jt messaging.";

    public HelloWorldMessage() {
    }

    // Attributes


    public void setGreetingMessage (String greetingMessage) {
        this.greetingMessage = greetingMessage;

    }

    public String getGreetingMessage () {
        return (greetingMessage);
    }
}
```

When completed, this simple component is supposed to process an input message and return a reply message ("Hello World … welcome to Jt messaging! "). This component contains only one attribute (`greetingMessage`). This attribute represents the reply message to be returned.

3.3. Creating framework components

JtFactory implements the factory method design pattern. JtFactory also implements the core functionality of the Jt framework: creating framework components, setting/getting attribute values and removing components. createObject() is the method used for creating framework components. The following section of code creates an instance of the HelloWorld class.

```
JtFactory factory = new JtFactory ();

// Create helloWorld (HelloWorldMessage class)

helloWorld = factory.createObject (HelloWorldMessage.JtCLASS_NAME);
```

createObject returns a reference to the component. It accepts one or two parameters: class name and component Id. The component Id is optional. It should only be used in especial circumstances. For instance, when the component needs to be accessed from a remote application. The following call creates an instance of the class HelloWorldMessage named “helloWorld”:

```
helloWorld = (HelloWorldMessage) factory.createObject(HelloWorldMessage.JtCLASS_NAME,
                                          "helloWorld");
```

The component Id or the object reference can now be used to access the object.

```
HelloWorldMessage helloWorld = new HelloWorldMessage ();
```

The line above is the standard way in which Java objects are created. It may be used as well. On the other hand, createObject has several advantages:

1) CreateObject takes full advantage of the framework logging capabilities. The createObject operation is logged. This is useful when debugging and troubleshooting framework applications.

2) CreateObject handles singletons. This applies to single and multithreaded applications.

3) Attribute values can be automatically loaded from a properties file or input stream.

4) CreateObject takes advantage of the factory design pattern.

5) A component Id can be associated with the component. This Id is added to the component registry. It can be used to gain access to the object. This is especially useful when dealing with remote componentts accessed via web services, EJBs or similar remote APIs. Attribute values associated with a specific component Id can be loaded from a properties file or input stream.

3.4. Setting/Getting attribute values

As explained before, framework components are basically Java beans. JtFactory provides a couple of methods to set/get the value of bean attributes: setValue and getValue. The following examples illustrate the use of these methods.

```
factory.setValue (helloWorld, "greetingMessage", "Hello World ...");
```

or

```
factory.setValue ("helloWorld", "greetingMessage", "Hello World ...");
```

These two calls set the value of the attribute greetingMessage to be "Hello World …".

```
String msg = (String) factory.getValue (helloWorld, "greetingMessage");
```

or

```
String msg = (String) factory.getValue ("helloWorld", "greetingMessage");
```

These two calls return the value of the attribute greetingMessage.

```
helloWorld.getGreetingMessage();
helloWorld.setGreetingMessage("Hello World ...");
```

The two calls above use the standard Java setter and getter. They are valid as well. On the other hand, getValue and setValue have several advantages:

1) The standard Java setters and getters cannot be used when dealing with remote components. On the other hand, getValue and setValue apply to remote components accessed via web services, EJBs or similar APIs. The Jt framework provides transparent access.

2) getValue and setValue take full advantage of the framework logging capabilities. These operations are logged. This is especially useful when debugging and troubleshooting framework applications.

In summary there are cases where the standard setter/getter can be used. There are other cases where getValue and setValue should be used because of their additional advantages.

The framework allows you to set/get attribute values by using the component Id or reference. The attribute value can use its String representation. If the attribute type is in the following list, setValue converts string values into the appropriate type. For instance, if the attribute type is int, “2” is converted to its integer representation (2).

A) byte

B) short

C) int

D) long

E) float

F) double

G) boolean

H) char

I) Date (DateFormat.SHORT – short date format for the current locale)

The following statements would be valid:

factory.setValue (component, “intAttributeName”, “2”); // “2” is converted to integer

factory.setValue (component, “floatAttributeName”, “2.0”); // “2.0” is converted to float

factory.setValue (component, “booleanAttributeName”, “true”); // “true” is converted to boolean

factory.setValue (component, “dateAttributeName”, “01/12/2008”); // short date in current locale is converted to Date

When an object reference is passed to setValue, no conversion is performed:

Date objReference = new Date ();

factory.setValue (component, “dateAttributeName”, objReference);

factory.getValue returns an object reference. This applies to primitive types as well (int, float, long, etc.):

Integer I = (Integer) factory.getValue (component, “intAttributeName”);

Float F = (Float) factory.getValue (component, “floatAttributeName”);

Boolean B = (Boolean) getValue (component, “booleanAttributeName”);

Date d = getValue (component, “dateAttributeName”);

3.5. Creating framework messages

Framework messages can use any format. No restrictions are imposed. JtInterface can handle any type of message (Object class). You can define your own message types; MyMessageType for instance. Framework components are instructed to perform operations by using these messages. Many of the core Jt components use JtMDPMessage. This class defines messaging information (Id, content, data, attachment, etc.). Instances of JtMDPMessage are created as follows:

```
// Create a Message.

JtMDPMessage msg = new JtMDPMessage ();
```

or

```
// Create a Message. Message ID is HelloWorld.JtHELLO

JtMDPMessage msg = new JtMDPMessage (HelloWorld.JtHELLO);
```

These examples create an instance of JtMDPMessage. The second example sets the message ID to be `HelloWorld.JtHELLO`. This message may be used to instruct the helloWorld object to return the greeting message.

Arbitrary attributes can be associated to the message:

```
    msg = new JtMDPMessage (JtComponent.JtCLONE);
    factory.setValue(msg, JtMDPMessage.OBJECT, object);
```

3.6. Sending framework messages

The sendMessage method (JtMessenger instance) is used to send messages to local and remote framework components:

```
messenger.sendMessage (helloWorld, msg);
```

or

```
messenger.sendMessage (“helloWorld”, msg);
```

Asynchronous and secure messaging are also supported:

```
messenger.setSynchronous(false);
messenger.setEncrypted(true);
```

3.7. Processing framework messages

processMessage is the most important method used by the Jt Framework. It is the single method declared by JtInterface This method is responsible for interpreting and processing framework messages. All framework components implement processMessage (JtInterface) or inherit it from JtObject (or any of its subclasses).

```
// Process object messages

public Object processMessage (Object message) {


    // Process the message

    if (message.equals("hi") || message.equals("hello"))
        return (greetingMessage);


    logger.handleError ("Invalid message:" + message);
    return (null);

}
```

processMessage can handle any type of message (Object class). This example uses String. When the message (“hi” or “hello”) is received, the greeting message (greetingMessage) is returned as the reply message. This is what our component is supposed to do.

3.8. Removing framework components

Jt components may need to be removed once they are not longer needed. removeObject is used for removing Jt components:

```
factory.removeObject (helloWorld);
```

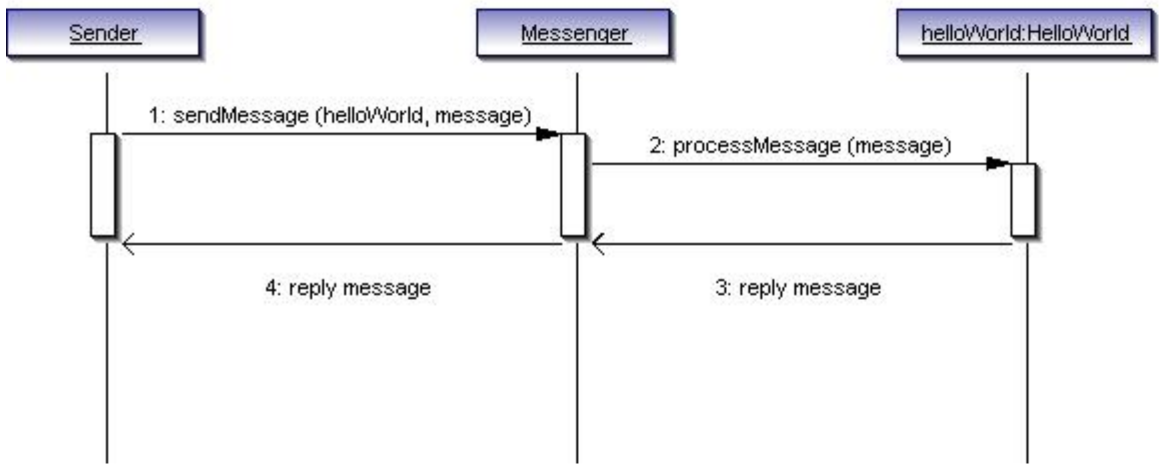
or

```
factory.removeObject ("helloWorld");
```

The object being removed is also removed from the component registry kept by the framework. At this stage, the object should be ready to be collected by the standard garbage collection mechanism.

removeObject needs one parameter: the Id or reference to the component to be removed. This method also applies to remote components accessed via proxies (web services, EJBs, etc).

3.9. Complete HelloWorld implementation



UML diagram of HelloWorld. A message is sent to the helloWorld component.

The following is the complete HelloWorldMessage implementation:

```
package Jt.examples;

import Jt.*;

/**
 * Demonstrates the Jt framework API.
 */

public class HelloWorldMessage extends JtObject {
    public static final String JtCLASS_NAME = HelloWorldMessage.class.getName();
    private JtLogger logger;
    private JtFactory factory = new JtFactory ();


    private static final long serialVersionUID = 1L;

    private String greetingMessage = "Hello World ... Welcome to Jt messaging!";

    public HelloWorldMessage() {
        logger = (JtLogger) factory.lookupObject(JtFactory.jtLogger);
    }

    // Attributes


    public void setGreetingMessage (String greetingMessage) {
        this.greetingMessage = greetingMessage;

    }

    public String getGreetingMessage () {
        return (greetingMessage);
    }
    // Process object messages

    public Object processMessage (Object message) {



        // Process the message

        if (message.equals("hi") || message.equals("hello"))
            return (greetingMessage);


        logger.handleError ("Invalid message:" + message);
        return (null);

    }


    /**
     * HelloWorld program. Demonstrates the use of the
     * framework messaging API.
     * 1) JtFactory creates an instance of HelloWorld.
     * 2) Sends a message to the new instance and prints the reply.
     */

    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();  // Jt Factory
        String reply;
        HelloWorldMessage helloWorld;
        JtMessenger messenger = new JtMessenger ();


        // Create helloWorld (HelloWorldMessage class)

        helloWorld = (HelloWorldMessage) factory.createObject (HelloWorldMessage.JtCLASS_NAME);

        // Send the Message

        System.out.println ("main:sending a message to the helloWorld object ...");
        reply = (String) messenger.sendMessage (helloWorld, "hi");

        // Print the reply message (Greeting)

        System.out.println (reply);



    }

}
```

The following is another valid version of HelloWorld. It uses the JtInterface instead of creating a subclass of JtObject. Integer constants are used as message Ids. Strings are often used as message Ids because the component may need to be added to BPM/UML/BPEL diagrams. Strings are more natural and easier to handle in this situation.

```
package Jt.examples;

import Jt.*;

/**
 * Demonstrates the Jt framework messaging API.
 */

public class JtHelloWorld implements JtInterface {
    public static final String JtCLASS_NAME = JtHelloWorld.class.getName();
    public static final int JtHELLO = 1;
    private JtFactory factory = new JtFactory ();
    private JtLogger logger;


    private static final long serialVersionUID = 1L;

    private String greetingMessage;

    public JtHelloWorld() {
       logger = (JtLogger) factory.lookupObject(JtFactory.jtLogger);
    }

    // Attributes


    public void setGreetingMessage (String greetingMessage) {
        this.greetingMessage = greetingMessage;

    }

    public String getGreetingMessage () {
        return (greetingMessage);
    }


    // Process object messages

    public Object processMessage (Object message) {

        Integer msgid = null;
        JtMDPMessage msg = (JtMDPMessage) message;


        if (msg == null)
            return null;


         msgid = (Integer) factory.getValue(message, JtMDPMessage.ID);

        if (msgid == null)
            return null;
        // Process the JtHELLO Message

        switch (msgid.intValue()) {
        case JtHELLO:

                if (greetingMessage == null)
                        greetingMessage = "Hello World ...";

                logger.handleTrace
                    ("HelloWorld returning a greeting message: " +  greetingMessage);

                return (greetingMessage);
        default:
                logger.handleError ("Invalid message Id: " +  msgid.intValue());

        }

        return (null);

    }


    /**
     * HelloWorld program. Demonstrates the use of the
     * framework messaging API.
     * 1) JtFactory creates an instance of HelloWorld.
     * 2) Sends a message to the new instance and prints the reply.
     */

    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();  // Jt Factory
        JtMessenger messenger = new JtMessenger ();
        String reply;
        JtHelloWorld helloWorld;


        // Create helloWorld (HelloWorld class)

        helloWorld = (JtHelloWorld) factory.createObject (JtHelloWorld.JtCLASS_NAME);


        // Create a Message ("JtHELLO")

        JtMDPMessage msg = new JtMDPMessage (JtHELLO);

        // Send the Message

        System.out.println ("main:sending a message (JtHello) to the helloWorld object ...");
        reply = (String) messenger.sendMessage (helloWorld, msg);

        // Print the reply message (Greeting)

        System.out.println (reply);


    }

}
```

3.10. Loading attribute values from a properties file

We have seen that framework components can be customized by setting their attribute values. This can be done by using the setValue method or the standard Java setters. In many cases, we don't want to hard code the attribute values. The Jt Framework supports the use of a properties file. This file needs to be in the classpath and named Jt.properties. Attribute values are loaded from this resource file immediately after the object is created. The same mechanism applies to all the components and APIs integrated with the Jt Framework: Enterprise Java Beans (EJBs), Web Services, BPM, JMS, JtLogger, etc. This provides a consistent way of customizing framework applications. The following is an example of how Jt.properties looks like:

! Jt resource file

HelloWorld.greetingMessage:Hi there

Each line in the resource file uses the syntax *className.attribute:value* or *#objectName.attribute:value*

When a component with the specified class or name is created, the attribute is initialized with the value found in the properties file. A complete example is shown below:

```
! Jt properties file

! Hello World demo application

!Jt.JtLogger.logging:true
!Jt.JtLogger.logLevel:0
!Jt.JtLogger.logFile:log.txt

Jt.examples.HelloWorld.greetingMessage:Hi there ...

! The attribute can also be initialized by using the object name instead of
! the class.
!#helloWorld.greetingMessage:Hi there1 ...



! JDBC adapter (MySQL settings)

Jt.JtJDBCAdapter.user:root
Jt.JtJDBCAdapter.password:123456
Jt.JtJDBCAdapter.driver:com.mysql.jdbc.Driver
Jt.JtJDBCAdapter.url:jdbc:mysql://localhost/test
!Jt.JtJDBCAdapter.datasource:datasrc


! JMS Adapter (point-to-point)

Jt.jms.JtJMSQueueAdapter.queue:testQueue
Jt.jms.JtJMSQueueAdapter.connectionFactory:TestJMSConnectionFactory
Jt.jms.JtJMSQueueAdapter.timeout:1


! JMS Adapter (publish/subscribe)

Jt.jms.JtJMSTopicAdapter.topic:jtTopic
Jt.jms.JtJMSTopicAdapter.connectionFactory:TestJMSConnectionFactory
Jt.jms.JtJMSTopicAdapter.timeout:1


! Web services adapter

Jt.axis.JtWebServicesAdapter.url:http://www.domain.com/axis/services/JtAxisSe
rvice


! EJB Adapter


! Service Locator (Weblogic settings)

Jt.ejb.JtServiceLocator.url:t3://localhost:7001
Jt.ejb.JtServiceLocator.factory:weblogic.jndi.WLInitialContextFactory
Jt.ejb.JtServiceLocator.user:weblogic
Jt.ejb.JtServiceLocator.password:weblogic

! Java Mail Adapter

Jt.JtMail.server:serve.mydomain.com
Jt.JtMail.username:user
Jt.JtMail.password:password
Jt.JtMail.port:587

! JNDI Adapter

Jt.jndi.JtJNDIAdapter.factory:weblogic.jndi.WLInitialContextFactory
Jt.jndi.JtJNDIAdapter.url:t3://localhost:7001
```

3.11. Logging and debugging capabilities

The Jt Framework provides built-in logging and debugging capabilities. Jt can be instructed to automatically log all the operations performed on remote and local framework components. This includes creating components, setting attributes, sending messages, loading attribute values from the properties files, etc. This feature allows the user to identify problems quickly and easily. In order to debug our HelloWorld class, type the following command:

```
Java –DLog=stderr HelloWorld
```

This –DLog flag forces the logging information to be sent to the screen. This information can also be sent to a log file (log.txt):

```
Java –DLog=log.txt HelloWorld
```

In addition to framework messages, we can add our own information to the log file. The Jt Logger component (JtLogger instance) provides this functionality:

```
logger.handleTrace (“sending a message (JtHello) to the helloWorld object …”);
```

Logging can be enabled programatically:

```
logger = (JtLogger) factory.lookupObject(JtFactory.jtLogger);
factory.setLogging(true);
```

The class attribute *logging* enables the logging functionality. The framework resource file can be used to enable it as well:

```
Jt.JtLogger.logging:true
```

or

```
Jt.JtLogger.logFile:log.txt
```

The second line instructs the framework to enable logging and log messages using a file (log.txt).

The framework handles logging levels via the class attribute logLevel. This attribute specifies what messages should be logged: only messages above the current logging level are logged. This feature gives the user additional control over the logging capabilities. The default logging level is 3:

```
// Logging constants (JtLogger)

private static int JtDEFAULT_LOG_LEVEL = 3;      // Default Logging level
private static int JtMIN_LOG_LEVEL = 0;          // Minimum Logging level
```

These constants were taken from Jt.JtLogger. The current logging level can be changed programatically:

```
 logger.setLogLevel(2);
```

handleTrace handles one or two parameters. The second parameter is optional. It specifies the logging level of the particular message.

```
logger.handleTrace ("this message is level 2.", 2);
```

This message will be sent to the log only if the current logging level is 2 or lower. When the second parameter is omitted, the message will use the default logging level (3):

```
logger.handleTrace ("this message is level 3.");
```

The call below would filter no messages. All messages would be logged:

```
 logger.setLogLevel(JtLogger.JtMIN_LOG_LEVEL);
```

A warning message on the other hand, is always logged:

```
logger.handleWarning ("this message is always logged.");
```

Errors and exceptions are always logged as well:

```
logger.handleError ("this is an error message.");

logger.handleException (exception);
```

As usual the current logging level can be set using the framework resource file:

```
Jt.JtLogger.logLevel:0
```

The user can customize the standard logging capabilities and/or provide a logger component to meet specific requirements. Any logging solution/API can be easily integrated with the Jt framework.

3.12. Handling Errors and Exceptions

The Jt framework provides a consistent way of handling and propagating errors and exceptions. The same mechanism applies to all the APIs integrated with the Jt Framework: Enterprise Java Beans (EJBs), Servlets, Web Services, BPM, JMS, etc. Two methods are provided:

public void handleError (String msg);

and

public void handleException (Throwable e);

These methods are inherited from the JtOject class. handleException should be called each time an exception is detected and needs to be propagaged to the calling component. handleException sends the exception message and the stack trace to the screen or log file. It also stores the exception using the objException attribute. This attribute is inherited from JtObject. The objException attribute can be retrieved (getValue) and used to verify if an exception has occurred while the object was processing a message.

handleError should be called each time an error is detected and needs to be propagated. This method generates a JtException that is handled by the handleException method. Per the description above, handleException stores the exception and sends the exception message to the screen or log file. Errors and exceptions are always logged. They cannot be filtered using the logging levels.

The Jt framework propagates exceptions among its components. This is very useful. For instance, an exception caused during a remote component invocation may need to be propagated all the way to the client graphical user interface. The Jt framework does most of the propagation work automatically. You may also need to have exceptions propagated. The following piece of code is usually employed for this purpose:

```java
private Exception propagateException (JtObject obj) {
      Exception ex;

      if (obj == null)
            return (null);
      ex = (Exception) obj.getObjException();

      if (ex != null)
            this.setObjException(ex);

      return (ex);
}
```

The user can customize the exception handling capabilities and/or provide an exception handler to meet specific requirements.

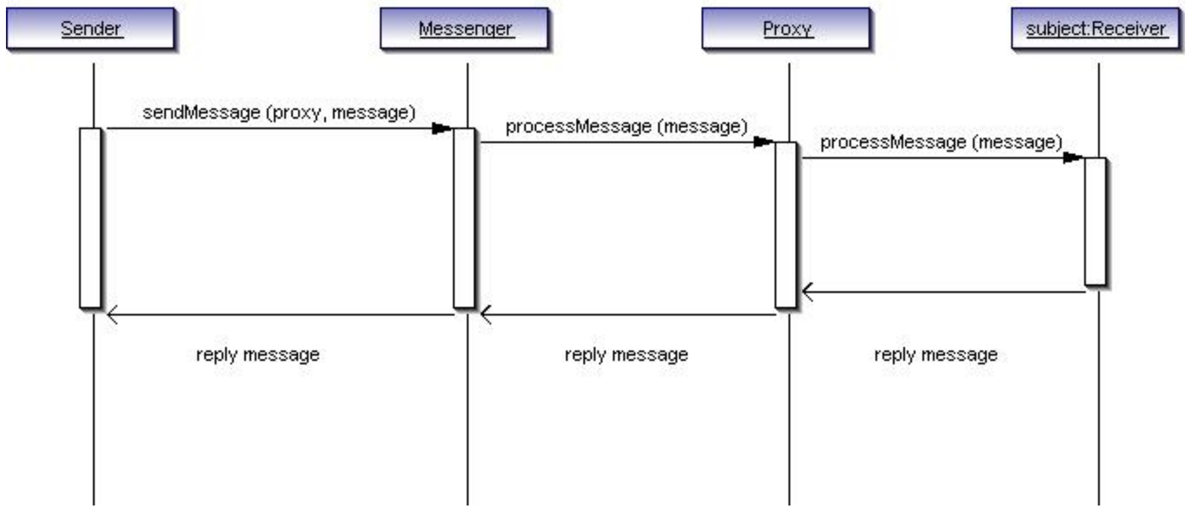
4. Design Pattern implementation using MDP

4.1. Gang of Four Design Patterns

Most design patterns have to deal with the interchange of information between pattern participants.The messaging design pattern (MDP) has been used to implement and/or facilitate the implementation of other well-known design patterns like Gang of Four design patterns (GoF), DAO, MVC, J2EE Design patterns, etc. MDP also provides a more natural and straightforward implementation. Although synchronous messaging is shown, MDP also supports asynchronous messaging. For a complete description of the API used by all these design patterns, please review the framework API documentation. Examples to illustrate the use of the design patterns can be found under src/Jt/examples/patterns. A complete list of examples can be found towards the end of this chapter.

4.1.1. Proxy

Within a messaging model, communication with a remote component can be achieved through a messenger, a proxy or via a combination of both. A proxy acts as the local placeholder (reference) to the remote component. MDP facilitates the implementation of Proxy. Under the messaging paradigm, Proxy is mainly responsible for forwarding the input message to the real subject (receiver).



Jt.JtProxy implements the proxy design pattern. The subject attribute represents the real subject. The example below is taken from Jt.JtProxy. A proxy for helloWorld is created. The framework makes extensive use of this design pattern. The implementation of Web Services, J2EE design patterns and ESB components rely on JtProxy. A proxy is used to interface with the remote component. This local proxy can be manipulated using the framework messaging API. Jt.http.JtHttpProxy, Jt.ejb.JtEJBProxy and Jt.axis.JtAxisProxy illustrate the use of JtProxy.

Conceptually it is simple to visualize sending a message to a local component:

messenger.sendMessage (MDPComponent, message);

The same concept applies to remote components:

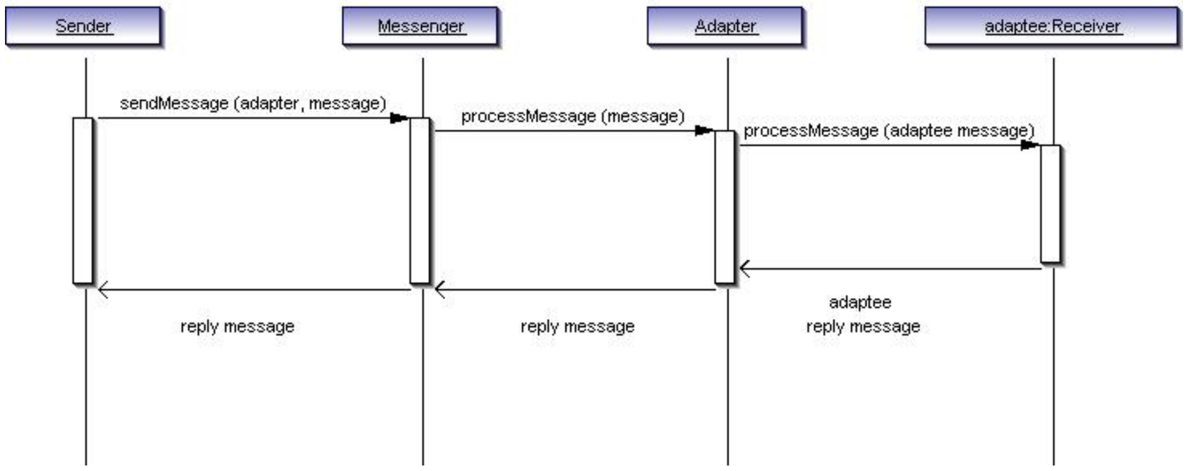
messenger.sendMessage (proxy, message);

A message can be sent to a remote MDP component using a messenger and a Proxy or reference to the remote component.

```java
public static void main(String[] args) {

  JtFactory factory = new JtFactory ();
  JtMessenger messenger = new JtMessenger ();
  HelloWorldMessage helloWorld;
  JtProxy proxy;
  String str;


  // Create an instance of JtProxy

  proxy = (JtProxy)
            factory.createObject (JtProxy.JtCLASS_NAME);
  helloWorld = (HelloWorldMessage) factory.createObject
                       (HelloWorldMessage.JtCLASS_NAME);

  // Initialize the proxy (subject attribute)
  proxy.setSubject(helloWorld); // this is the correct way of
                                // setting the local attribute
                                // subject


  // Update the attribute (that belongs to the subject) via its proxy

  factory.setValue(proxy, "greetingMessage", "hello world via a proxy");

  str = (String)factory.getValue(proxy, "greetingMessage");

  System.out.println ("greeting:" + str);

  // Send a message to the subject via its Proxy

  System.out.println ("Reply:" + messenger.sendMessage (proxy, "hi"));

  factory.removeObject(proxy);

}
```

4.1.2. Adapter

Under messaging and MDP, the main purpose of Adapter becomes the transformation of messages between message sender and receiver so that these components can be interconnected. For instance, you may need to implement a HTTP Adapter so that your local component can communicate with a remote component via the HTTP protocol. The same principle applies to other communication technologies and protocols



Jt.JtAdapter implements the adapter design pattern. The framework makes extensive use of this design pattern. Jt implements adapters for several APIs:

JDBC Adapter: Jt.JtDBCAdapter

HTTP Adapter: Jt.http.JtHttpAdapter

JMS Adapters: Jt.jms.JtJMSQueueAdapter and Jt.jms.JtJMSTopicAdapter

EJB Adapter: Jt.ejb.JtEJBAdapter

Web Services Adapter (Axis): Jt.axis.JtAxisAdapter

DAO Hibernate Adapter: Jt.hibernate.JtHibernateAdapter

Java Mail Adapter: Jt.JtMail

JNDI Adapter: Jt.jndi.JtJNDIAdapter

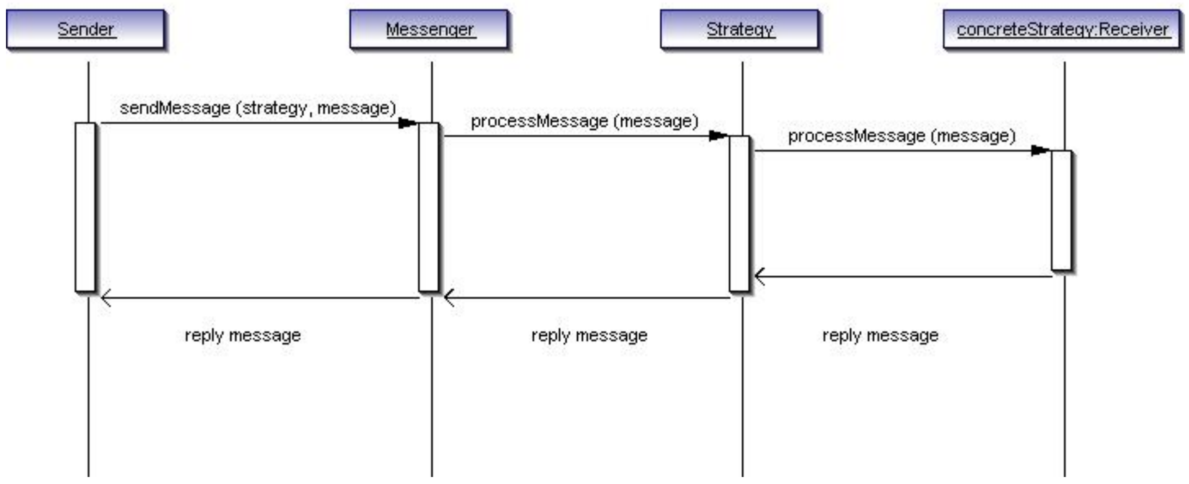
DOM Adapter: Jt.xml.JtDOMAdapter

Enterprise Service Bus Adapter:Jt.esb.JtESBAdapter

The Jt framework can be easily extended using additional adapters to handle others protocols and APIs. The source code of the above adapters demonstrates the use of JtAdapter. These adapters translate framework messages into calls understood by the appropriate API. This allows the external API to be plugged into framework applications. For instance, each one of these adapters can be added to a BPEL/BPM business process as part of complex applications.

4.1.3. Strategy

Under a messaging paradigm, Strategy is mainly responsible for forwarding the message to the component that implements the concrete strategy.

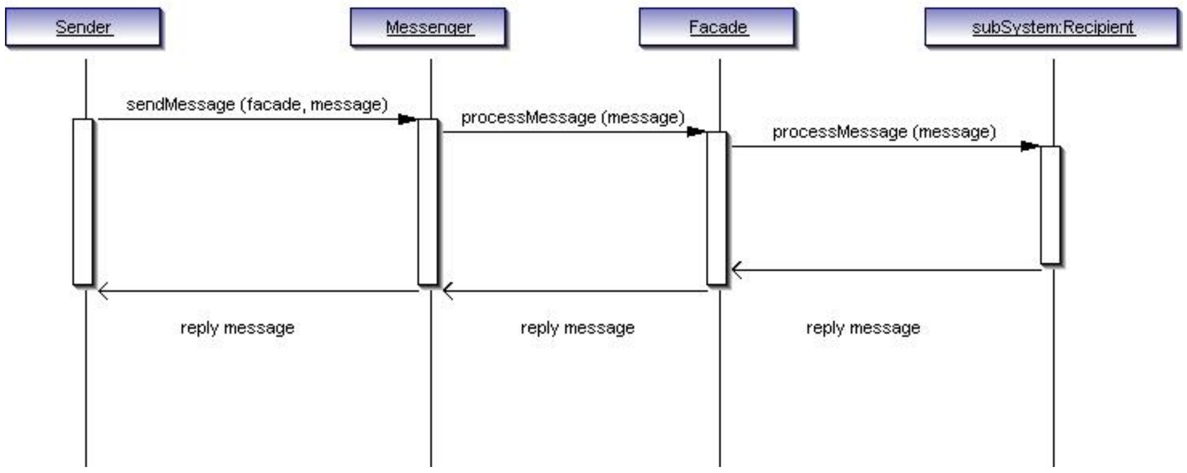


Jt.JtStrategy implements the strategy design pattern. The following section is taken from Jt.examples.patterns.StrategyExample. The concreteStrategy attribute defines the strategy to be employed. The framework supports several DAO strategies via Jt.DAO.JtDAOStrategy, including Hibernate and a native DAO implementation. Jt.DAO.JtDAOStrategy uses the strategy design pattern.

```
/**
 * Demonstrates the use of JtStrategy. It encodes
 * an object using two XML strategies.
 */

public static void main(String[] args) {

  JtFactory factory = new JtFactory ();
  JtMessenger messenger = new JtMessenger ();
  JtStrategy strategy;
  JtXMLHelper concreteStrategy;
  JtMDPMessage msg;
  HelloWorld hello = new HelloWorld ();

  hello.setGreetingMessage("Hello World ...");

  // Create an instance of JtStrategy

  strategy = (JtStrategy) factory.createObject (JtStrategy.JtCLASS_NAME);

  // Specify the concrete strategy to be executed.

  concreteStrategy = (JtXMLHelper) factory.createObject
                                          (JtXMLHelper.JtCLASS_NAME);

  strategy.setConcreteStrategy(concreteStrategy);

  // Encode the object using the XML strategy selected

  msg = new JtMDPMessage (JtComponent.JtXML_ENCODE);
  factory.setValue(msg, JtMDPMessage.OBJECT, hello);

  System.out.println
            ("Jt encoding:" +  messenger.sendMessage (strategy, msg));

  // Use a different encoding strategy

  strategy.setConcreteStrategy(new JtFactory ());
  System.out.println
             ("Java encoding:" + messenger.sendMessage (strategy, msg));


}
```

4.1.4. Façade

Under messaging and MDP, Façade is mainly responsible for forwarding the message to the appropriate subsystem.



Jt.Façade implements the façade design pattern. Jt.examples.patterns.TestFacade and Jt.RemoteFaçade demonstrate the use of this pattern. Jt.RemoteFaçade is part of the framework web services, J2EE pattern, and ESB implementations.

```
    /**
     * Implements a testing Facade for the Jt framework
     */

    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();
        JtFacade facade;
        Boolean status;

        // Create an instance of JtFacade

        facade = (JtFacade) factory.createObject (TestFacade.JtCLASS_NAME);

        // Test Jt modules using a Facade

        status = (Boolean) messenger.sendMessage (facade,
                                 new JtMessage (JtObject.JtTEST));

        if (!status.booleanValue()) {
            System.out.println ("TestFacade: one of more tests failed");
            System.exit (1);
        }


    }
```

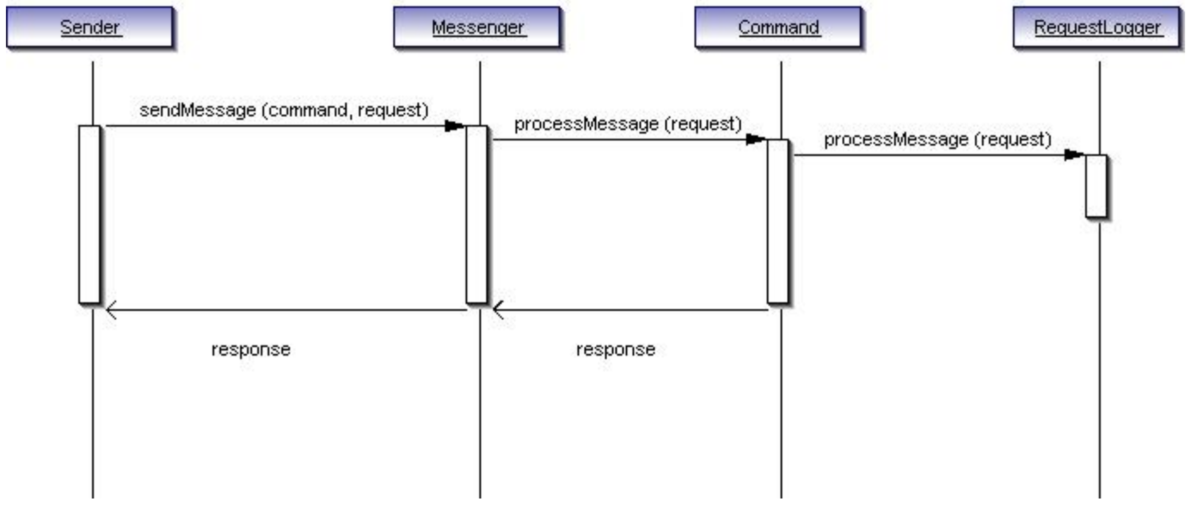
4.1.5. Memento

Jt.JtMemento implements the memento design pattern. JtMemento relies on XML in order to save and restore the state of an object. Please refer to the XML integration section of this document. Two messages are processed by the Jt.JtMemento class: JtSAVE & JtRESTORE. The message contains reference to the object to be saved or restored (JtMDPMesssage.*OBJECT*). The following section of code saves and restores the state of an instance of HelloWorld. The JtPortal reference application makes use of this design pattern. For instance, Jt.portal.Password and Jt.xml.JtXMLMemento.

```
/**
   * Demonstrates the messages processed by JtMemento.
   */

 public static void main(String[] args) {

   JtFactory factory = new JtFactory ();
   JtMessenger messenger = new JtMessenger ();
   JtMemento memento;
   JtPrinter printer = new JtPrinter ();
   JtMDPMessage msg;
   HelloWorld hello = new HelloWorld ();

   hello.setGreetingMessage("Hello World");

   // Create an instance of JtMemento
   memento = (JtMemento) factory.createObject (JtMemento.JtCLASS_NAME);

   // Save the current state of hello using Memento
   msg = new JtMDPMessage (JtMemento.JtSAVE);
   factory.setValue(msg, JtMDPMessage.OBJECT, hello);

   messenger.sendMessage (memento, msg);

   System.out.println ("Saved object:");
   printer.processMessage(hello);

   hello.setGreetingMessage("new message");

   System.out.println ("Object after change:");

   printer.processMessage(hello);

   // Restore the object

   msg = new JtMDPMessage (JtMemento.JtRESTORE);
   factory.setValue(msg, JtMDPMessage.OBJECT, hello);
   messenger.sendMessage (memento, msg);

   System.out.println ("Restored Object:");
   printer.processMessage(hello);

 }
```

4.1.6. Command

MDP facilitates the implementation of Command. Under the messaging paradigm, Command is responsible for processing the request. It may also queue or log requests (RequestLogger).



Jt.JtCommand implements the command design pattern. This implementation features logging and undo capabilities. JtCommand can also execute using a separate/independent thread and process requests via a message queue. JtCommand inherits from JtAnimatedComponent. The following section of code taken from Jt.examples.patterns.BankAccount illustrates the use of JtCommand. A few basic account operations are shown. This example handles a request log. The logMessage() method is inherited from JtCommand. It allows us to log the requests. The request log can be retrieved via the “messageLog” attribute of JtCommand.

```
   /**
     * Bank Account implementation (main)
     */

    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();
        JtMDPMessage msg;
        BankAccount account;

        // Create a BankAccount

        account = (BankAccount) factory.createObject
                                 (BankAccount.JtCLASS_NAME);


        // Perform some transactions (operations on the account)

        System.out.println ("Deposit: 500.1");
        msg =  new JtMDPMessage (BankAccount.DEPOSIT);
        factory.setValue (msg, "amount", new  Double (500.1));

        messenger.sendMessage (account, msg);
        System.out.println ("Balance:" + account.getBalance ());

        System.out.println ("Withdrawal: 200.1");
        msg =  new JtMDPMessage (BankAccount.WITHDRAWAL);
        factory.setValue (msg, "amount", new  Double (200.1));

        messenger.sendMessage (account, msg);
        System.out.println ("Balance:" + account.getBalance ());

        // Print the log (list of transactions)

        System.out.println ("Transaction log:\n");
        messenger.sendMessage (account,
                  new JtMDPMessage (BankAccount.PRINT_TRANSACTIONS));



    }

}
```

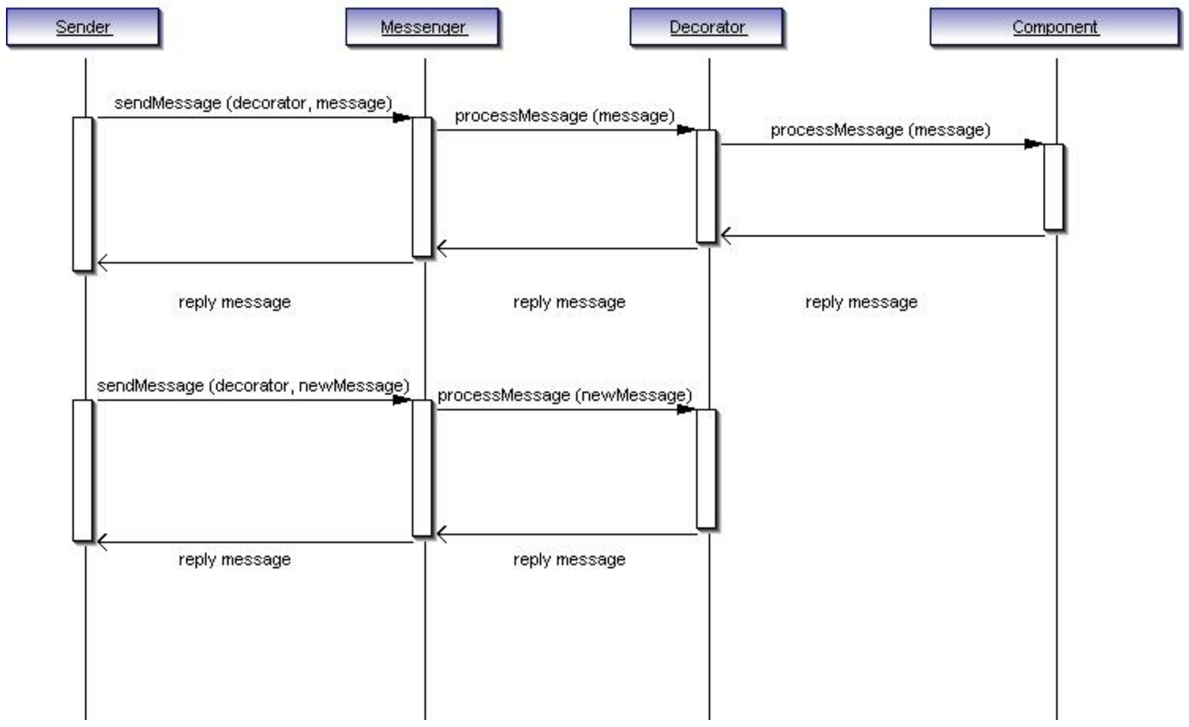
4.1.7. Composite

Jt.JtComposite implements the composite design pattern. Messages include JtADD_CHILD, JtREMOVE_CHILD and JtGET_CHILD. The JtPortal classes Jt.portal.Checkout and Jt.portal.ShoppingCart illustrate the use of JtComposite.

```
     /**
      * Demonstrates the implementation of the Composite design
      * pattern via JtFactory
      */

  public static void main(String[] args) {

    JtFactory factory = new JtFactory ();
    JtComposite composite, composite1;
    Double leaf1 = new Double (1.0);
    Double leaf3 = new Double (3.0);
    JtMDPMessage msg;
    JtPrinter printer = new JtPrinter ();

    // Create an instance of JtComposite

    composite = (JtComposite) factory.createObject
                          (JtComposite.JtCLASS_NAME);

    // Add objects to the composite (using JtFactory)

    msg = new JtMDPMessage (JtComponent.JtADD_CHILD);
    factory.setValue(msg, JtMDPMessage.OBJECT, composite);
    factory.setValue(msg, JtMDPMessage.CHILD, leaf1);

    factory.processMessage(msg);

    factory.setValue(msg, JtMDPMessage.OBJECT, composite);
    factory.setValue(msg, JtMDPMessage.CHILD, leaf3);
    factory.processMessage(msg);

    msg = new JtMDPMessage (JtComposite.JtREMOVE_CHILD);
    factory.setValue(msg, JtMDPMessage.OBJECT, composite);
    factory.setValue(msg, JtMDPMessage.CHILD, new Double (3.0));
    factory.processMessage(msg);

    msg = new JtMDPMessage (JtComponent.JtGET_CHILD);
    factory.setValue(msg, JtMDPMessage.OBJECT, composite);
    factory.setValue(msg, JtMDPMessage.CHILD, new Double (3.0));

    // Print the composite (XML format)

    printer.processMessage(composite) ;

  }
```

4.1.8. Decorator

When MDP is used, Decorator is responsible for implementing new functionality. Decorator is also responsible for forwarding messages (related to the existing functionality) to the decorated component.



Jt.JtDecorator implements the decorator design pattern. The following example is extracted from Jt.examples.pattern.DecoratedHelloWorld.

```
// Demonstrates the use of JtDecorator.
// This version of HelloWorld is able to handle an
// additional language.

public static void main(String[] args) {

  JtFactory factory = new JtFactory ();
  String reply;
  JtDecorator decorator;
  HelloWorld helloWorld = new HelloWorld ();
  JtMessenger messenger = new JtMessenger ();

  // Create helloWorld (DecoratedHelloWorld class)

  decorator = (JtDecorator) factory.createObject

                      (DecoratedHelloWorld.JtCLASS_NAME);
  decorator.setComponent(helloWorld) ; // Component to be decorated

  // Send the Message

  reply = (String) messenger.sendMessage
               (decorator, new JtMDPMessage (HelloWorld.JtHELLO));

  // Print the reply message (Greeting)

  System.out.println (reply);

  // Try the new functionality provided by JtHOLA

  reply = (String) messenger.sendMessage
            (decorator, new JtMDPMessage (DecoratedHelloWorld.JtHOLA));

  // Print the reply message
  System.out.println (reply);



}
```

4.1.9. Prototype

The prototype design pattern is implemented via the JtCLONE message and JtFactory. In general, any Java component can be cloned (some exceptions apply):

```
    msg = new JtMDPMessage (JtObject.JtCLONE);
    factory.setValue(msg, JtMDPMessage.OBJECT, obj);
    Object clone = factory.processMessage(obj);
```

JtFactory is responsible for implementing the ***clone*** functionality/message: it returns a copy of the object when the JtCLONE message is received.

```
  public static void main(String[] args) {
    JtFactory factory = new JtFactory ();
    JtComposite aux;
    JtPrototype tree, branch;
    Double leaf1 = new Double (1.0);
    Double leaf2 = new Double (2.0);
    Double leaf3 = new Double (3.0);
    JtMessage msg;
    JtPrinter printer = new JtPrinter ();

    // Create an instance of JtPrototype

    tree = (JtPrototype) factory.createObject (JtComposite.JtCLASS_NAME);

    // Add objects to the tree

    msg = new JtMessage (JtComposite.JtADD_CHILD);
    msg.setMsgContent(leaf1);

    factory.sendMessage(tree, msg);
    msg.setMsgContent(leaf2);
    factory.sendMessage(tree, msg);

    branch = (JtComposite) factory.createObject (JtComposite.JtCLASS_NAME);
    msg.setMsgContent(leaf3);

    factory.sendMessage(branch, msg);

    msg = new JtMessage (JtComposite.JtADD_CHILD);
    msg.setMsgContent(branch);
    factory.sendMessage(tree, msg);

    // Clone the object. In this case the tree is cloned
    aux = (JtComposite) factory.sendMessage(tree,
       new JtMessage (JtObject.JtCLONE));

    msg = new JtMDPMessage (JtObject.JtCLONE);
    factory.setValue(msg, JtMDPMessage.OBJECT, tree);
    JtComposite composite3 = (JtComposite) factory.processMessage(msg);
  }
```

4.1.10. Iterator

Jt.JtIterator implements the iterator design pattern. JtNEXT returns the next element in the iteration or null if the end has been reached. The following example iterates over the items of a collection using JtIterator. JtList and JtCollection can be traversed using an iterator.

```
    public static void main(String[] args) {

      JtFactory factory = new JtFactory ();
      JtMessenger messenger = new JtMessenger ();
      JtMDPMessage msg;
      JtIterator it;
      Integer intObj;
      JtCollection collection;

      // Create a JtColletion

      collection = (JtCollection)
              factory.createObject (JtCollection.JtCLASS_NAME);

      msg = new JtMDPMessage (JtComponent.JtADD);

      // Add objects to the collection

      factory.setValue(msg, JtMDPMessage.OBJECT, new Integer(1));
      messenger.sendMessage (collection, msg);

      factory.setValue(msg, JtMDPMessage.OBJECT, new Integer(2));
      messenger.sendMessage (collection, msg);
      // Retrieve the iterator associated with the collection
      it = collection.getIterator();
      // Traverse the collection using the iterator (JtNEXT message)

      while ((intObj = (Integer) messenger.sendMessage (it,
                new JtMDPMessage (JtIterator.JtNEXT))) != null) {
            System.out.println ("Object=" + intObj);

            messenger.sendMessage (it,
                      new JtMDPMessage (JtIterator.JtREMOVE_CURRENT));

      }
      it = collection.getIterator();
      while ((intObj = (Integer) messenger.sendMessage (it,
            new JtMDPMessage (JtIterator.JtNEXT))) != null)
            System.out.println ("Object=" + intObj);

    }
```

4.1.11. Flyweight

Jt.JtFlyweight implements the flyweight design pattern. JtFlyweight extends JtComposite. The factory attribute specifies the flyweight factory. JtFlyWeight handles two messages: JtCREATE_FLYWEIGHT & JtGET_FLYWEIGHT. JtGET_FLYWEIGHT returns the flyweight. If the flyweight doesn't exist, it gets created by sending a JtCREATE_FLYWEIGHT message to the flyweight factory. The following piece of code, extracted from JtFlyweight, processes these messages:

```
    public Object processMessage (Object message) {

      Object msgid = null;
      JtMessage e = (JtMessage) message;
      JtMessage tmp;
      Object aux, aux1;
      JtMDPMessage msg;
      JtMessenger messenger = new JtMessenger ();
      if (e == null)
          return null;

      msgid = (String) e.getMsgId ();

      if (msgid == null)
          return null;
      if (msgid.equals (JtFlyweight.JtGET_FLYWEIGHT)) {


          msg = new JtMDPMessage (JtComposite.JtGET_CHILD);
          factory1.setValue(msg, JtMDPMessage.OBJECT, this);
          factory1.setValue(msg, JtMDPMessage.CHILD, e.getMsgContent ());

          aux = factory1.processMessage (msg);

          if (aux != null)
              return (aux);

          if (factory == null) {
              logger.handleError ("processMessage: factory attribute needs to be set");
              return (null);
          }
          logger.handleTrace ("JtFlyweight: processMessage creating a new flyweight");

          tmp = new JtMessage (JtFlyweight.JtCREATE_FLYWEIGHT);
          tmp.setMsgContent (e.getMsgContent ());

          aux1 = messenger.sendMessage (factory, tmp);

          msg = new JtMDPMessage (JtComposite.JtADD_CHILD);

          factory1.setValue(msg, JtMDPMessage.OBJECT, this);
          factory1.setValue(msg, JtMDPMessage.CHILD, aux1);

          factory1.processMessage(msg);

          return (aux1);
      }
      logger.handleError("Invalid message ID:" + msgid);

      return (null);
  }
```

4.1.12. Observer

Jt.JtObservable implements the observer design pattern. JtObservable inherits functionality from several pattern classes including JtComposite and JtThread. Instances of this class can run using an independent thread. This class handles 3 types of messages: JtADD_OBSERVER (adds an observer), JtNOTIFY_OBSERVERS (notify observers) and JtREMOVE_OBSERVER (remove observer). The notification message to be sent to the observers can be specified as part of JtNOTIFY_OBSERVERS.

Jt.examples.patterns.Alarm demonstrates the use of JtObservable. The user is notified after a predetermined amount of time. This implementation sends a notification to the observer after the time period expires. A dialog pops up.

```
    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();
        JtMDPMessage msg;
        JtDialog dialog = new JtDialog (); // Dialog box
        Alarm alarm;


        // Create the alarm component

        alarm = (Alarm) factory.createObject (Alarm.JtCLASS_NAME);

        alarm.setdaemon(false); // user thread. It will
                                // keep running after this program exits.

        // Add an observer (dialog component). A dialog should pop up once
        // the time is up.

        msg = new JtMDPMessage (JtObservable.JtADD_OBSERVER);
        factory.setValue(msg, JtMDPMessage.OBJECT, dialog);
        dialog.setMessage("Time is up ....");

        messenger.sendMessage (alarm, msg);

        alarm.setTime(4000L);   // Notify the observer(s) after 4 seconds

        // Activate the alarm component

        messenger.setSynchronous(false);
        messenger.sendMessage (alarm,
                new JtMDPMessage (JtObject.JtACTIVATE));


    }
```

4.1.13. Template Method

Jt.JtTemplateMethod implements the template method design pattern. This is an abstract class. Jt.examples.patterns.TemplateMethodExample illustrates the use of this design pattern. The processMessage method is declared abstract:

```
/**
 * Jt Implementation of the Template Method pattern.
 */


abstract public class JtTemplateMethod extends JtObject {



  public JtTemplateMethod() {
  }



  /**
    * Defines processMessage as an abstract method. The actual
    * implementation of processMessage is left to the subclasses.
    *
    * @param message Jt Message
    */


  public abstract Object processMessage (Object message);


}
```

4.1.14. Chain of Reponsibility

Jt.JtChainOfResponsibility implements the chain of responsibility design pattern. Jt.examples.patterns.MultiHelloWorld shows an example. This class implements HelloWorld in multiple languages using chain of responsibility.

```
/**
 * Demontrates the use of JtChainOfResposibility. This version of
 * HelloWorld is able to handle additional languages.
 */


public static void main(String[] args) {

  JtFactory factory = new JtFactory ();
  JtChainOfResponsibility chain;
  JtMDPMessage msg;
  JtMessenger messenger = new JtMessenger ();


  // Creates an instance of the chain of responsibility

  chain = (JtChainOfResponsibility) factory.createObject

           (JtChainOfResponsibility.JtCLASS_NAME);


  // Add two components to the chain (one per each language).

  msg = new JtMDPMessage (JtComponent.JtADD_CHILD);
  factory.setValue (msg, JtMDPMessage.OBJECT, chain);
  factory.setValue (msg, JtMDPMessage.CHILD, new MultiHelloWorld ());
  factory.processMessage(msg);

  msg = new JtMDPMessage (JtComponent.JtADD_CHILD);
  factory.setValue (msg, JtMDPMessage.OBJECT, chain);
  factory.setValue (msg, JtMDPMessage.CHILD, new FrenchHelloWorld ());
  factory.processMessage(msg);


  // Send a message to the chain. The appropriate component
  // will process the request

  messenger.sendMessage (chain,
                 new JtMDPMessage (MultiHelloWorld.JtHELLO));

  messenger.sendMessage (chain,
                 new JtMDPMessage (FrenchHelloWorld.JtBONJOUR));


}
```

4.1.15. Builder

Jt.JtBuilder implements the builder design pattern. Jt.examples.patterns.BuiderExample shows an example:

```
  /**
    * Demonstrates the use of the JtBuilder design pattern.
    */

   public static void main(String[] args) {

       JtFactory factory = new JtFactory ();
       JtMessenger messenger = new JtMessenger ();
       JtBuilder builder;
       Object expression;
       JtPrinter printer = new JtPrinter ();


       // Create an instance of JtBuilder

       builder = (JtBuilder) factory.createObject (JtBuilder.JtCLASS_NAME);
       builder.setConcreteBuilder(new BuilderExample ());

       // Build the expression using a composite (tree representation)

       expression = (JtInterface) messenger.sendMessage (builder,
                                    new JtMDPMessage (JtBuilder.JtBUILD));


       printer.processMessage(expression);

       // Build the expression using another concrete builder (list
       // representation)

       builder.setConcreteBuilder(new BuilderExample1 ());

       expression =  messenger.sendMessage (builder,
                                      new JtMessage (JtBuilder.JtBUILD));

       printer.processMessage(expression);


   }
```

4.1.16. Abstract Factory

Jt.JtAbstractFactory implements the abstract factory design pattern. Jt.examples.patterns.SwitchFactory is an example of the use of this design pattern.

```
    /**
     * Process object messages.
     */

    public Object processMessage (Object message) {

        Object msgid = null;
        JtMessage e = (JtMessage) message;

        String data;
        Switch output;
        JtFactory factory = new JtFactory ();


        if (e == null)
            return null;

        msgid = (String) e.getMsgId ();

        if (msgid == null)
            return null;


        if (msgid.equals (JtFlyweight.JtCREATE_FLYWEIGHT)) {
            data = (String) e.getMsgContent ();
            if (data == null) {
                logger.handleError ("JtCREATE_FLYWEIGHT: invalid key");
                return (null);
            }

            if (data.equals ("On")) {

                output = (Switch) factory.createObject (Switch.JtCLASS_NAME, "On");
                output.setConcreteState(new OnSwitch ());
            } else {
                output = (Switch) factory.createObject (Switch.JtCLASS_NAME, "Off");
                output.setConcreteState(new OffSwitch ());

            }
            return (output);
        }


        logger.handleError("Invalid message ID:" + msgid);
        return (null);


    }
```

4.1.17. Factory Method

Jt.JtFactoryMethod implements the factory method design pattern. Jt.JtFactory extends Jt.JtFactoryMethod. It demonstrates the use of this design pattern.

4.1.18. Singleton

Jt.JtSingleton and Jt.JtSingletonComponent implement the singleton design pattern. Framework singletons may extend these classes. Jt.examples.SingletonHelloWorld demonstrates the use of this design pattern. The createObject method needs to be used in order to create and handle framework singletons. The first invocation creates the singlenton. Additional calls to createObject do not create new instances. The same instance is returned instead. The JtSingleton implementation is able to handle multithreaded applications. Several framework and JtPortal classes use the singleton design pattern. For instance Jt.DAO.JtDAOContext (Jt DAO implementation) and Jt.chat.ChatManager.

You don't have to subclass Jt.JtSingleton to use singletons.There is a second option. JtFactory allows you to create singletons. The attribute createSingleton needs to be true before createObject is invoked. Jt.servlet.JtServlet and Jt.axis.JtAxisService show this approach.

```
/**
 * Demonstrates the messages processed by JtSingleton.
 * It also demonstrates the use of singletons via JtFactory.
 */

public static void main(String[] args) {

  JtFactory factory = new JtFactory ();
  JtSingleton singleton, singleton1;
  JtObject obj, obj1;

  System.out.println ("Creating an instance of a singleton ...");

  // Create a JtSingleton instance

  singleton = (JtSingleton) factory.createObject (JtSingleton.JtCLASS_NAME, "singleton");


  System.out.println ("Attempting to create a second instance of a singleton ...");

  singleton1 = (JtSingleton) factory.createObject (JtSingleton.JtCLASS_NAME, "singleton1");

  if (singleton == singleton1)
      System.out.println("create singleton (via inheritance): GO");
  else
      System.out.println("create singleton (via inheritance): FAIL");


  // Demonstrate the handling of singletons (via JtFactory)
  // Create the singlenton (via factory)

  factory.setCreateSingleton (true);
  obj = (JtObject) factory.createObject(JtObject.JtCLASS_NAME);

  // The singleton instance is returned. No new instance is created

  factory.setCreateSingleton (true);
  obj1 = (JtObject) factory.createObject(JtObject.JtCLASS_NAME);

  if (obj == obj1)
      System.out.println("create singleton (via JtFactory): GO");
  else
      System.out.println("create singleton (via JtFactory): FAIL");

}
```

4.1.19. Interpreter

Jt.JtIntepreter implements the intepreter design pattern.

```
/**
  * Process object messages. Subclasses need to override this method and
  * implement the JtINTERPRET message.
  */

public Object processMessage (Object message) {

    Object msgid = null;
    JtMDPMessage e;
    JtFactory factory = new JtFactory ();



    if (message == null)
        return null;

    if (message instanceof JtMDPMessage) {
      e = (JtMDPMessage) message;
      msgid = factory.getValue(message, JtMDPMessage.ID);


      if (msgid == null)
            return null;

      if (msgid.equals (JtInterpreter.JtINTERPRET)) {
            // subclass needs to implement/process this message
            return (null);
      }
    }
    return (null);
}
```

4.1.20. Mediator

Jt.JtMediator implements the mediator design pattern. JtMediator extends JtComposite. Jt.examples.patterns.Mediator demostrates the use of this design pattern:

```
    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();

        JtMediator mediator;
        Colleague colleague1, colleague2, colleague3;
        JtMessenger messenger = new JtMessenger ();

        System.out.println
                 ("Press any key to start/stop the chat room demo ...");
        waitForInputKey ();

        // Create an instance of the chat room

        mediator = (JtMediator) factory.createObject (Mediator.JtCLASS_NAME, "mediator");

        colleague1 = (Colleague) factory.createObject (Colleague.JtCLASS_NAME, "Jenny");
        factory.setValue (colleague1, "greetingMessage",
                      "Hi folks! I'm Jenny. How are you all doing ?");

        // Use asyncronous messaging
        messenger.setSynchronous(false);

        // Activate the first member (Animated Component)

        factory.setValue (colleague1, "mediator", mediator);
        messenger.sendMessage (colleague1, new JtMDPMessage (JtComponent.JtACTIVATE));


        // Activate the second member

        colleague2 = (Colleague) factory.createObject (Colleague.JtCLASS_NAME, "Daniel");

        factory.setValue (colleague2, "mediator", mediator);
        factory.setValue (colleague2, "greetingMessage", "Hi, I'm Daniel.");
        messenger.sendMessage (colleague2, new JtMDPMessage (JtComponent.JtACTIVATE));


        // Activate the third member

        colleague3 = (Colleague) factory.createObject (Colleague.JtCLASS_NAME, "Mary");

        factory.setValue (colleague3, "mediator", mediator);
        factory.setValue (colleague3, "greetingMessage", "Hi, I'm Mary.");
        messenger.sendMessage (colleague3, new JtMDPMessage (JtComponent.JtACTIVATE));


        waitForInputKey ();

        // Stop the Animated components (independent thread will stop).

        messenger.sendMessage (colleague1, new JtMDPMessage (JtComponent.JtSTOP));
        messenger.sendMessage (colleague2, new JtMDPMessage (JtComponent.JtSTOP));
        messenger.sendMessage (colleague3, new JtMDPMessage (JtComponent.JtSTOP));

        messenger.sendMessage (mediator, new JtMDPMessage (JtComponent.JtSTOP));

    }
```

4.1.21. Visitor

The MDP implementation of the visitor design pattern requires two messages: JtACCEPT and JtVISIT. Any framework component can implement the visitor design pattern by implementing these two messages. The visitor component is passed as part of the JtACCEPT message:

```
msgid = factory.getValue(message, JtMDPMessage.ID);
visitor = factory.getValue(message, JtMDPMessage.OBJECT);

if (msgid.equals (JtObject.JtACCEPT)) {

        if (visitor == null) {
                logger.handleError
                   ("Invalid JtACCEPT message. Visitor reference is null.");
                return (null);
        }

        // Send a JtVISIT message to the visitor.
        // Include a reference to this object

        msg = new JtMDPMessage (JtVisitor.JtVISIT);
        factory.setValue(msg, JtMDPMessage.OBJECT, this);

        return (messenger.sendMessage (visitor, msg));
}
```

The JtVISIT message contains a reference to the component to be visited (visitable). The visitor class needs to implement the JtVISIT message:

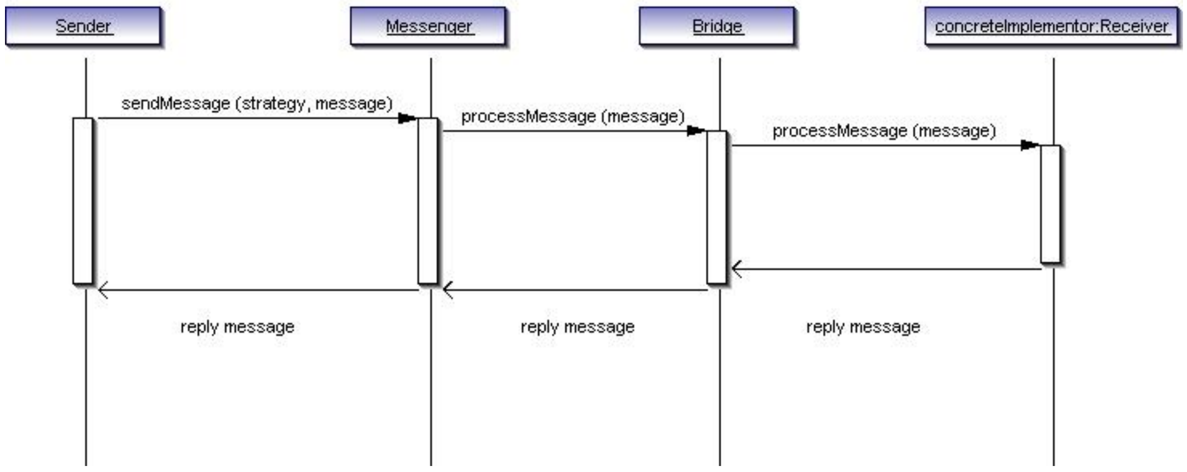
```
visitable = factory.getValue(message, JtMDPMessage.OBJECT);

if (msgid.equals (JtObject.JtVISIT)) {

       if (visitable == null) {
              logger.handleError ("Invalid message (JtVISIT)");
              return (null);
       }

    // Visit the component
    // Do something with the visitable here.

    return (null);
}
```

The following section of code is part of Jt.examples.patterns.Visitor. This class illustrates the processing of JtVISIT.

```
// Send a JtACCEPT message to the node. The message contains
// a reference to the visitor

msg = new JtMDPMessage (JtObject.JtACCEPT);
factory.setValue(msg, JtMDPMessage.OBJECT, visitor);
messenger.sendMessage (node, msg);
```

4.1.22. Bridge

When MDP is used, Bridge is mainly responsible for forwarding the message to the concrete implementor.



Jt.JtBridge implements the bridge design pattern.

```
/**
 * Example of the use of JtBridge.
 */

public class BridgeExample  {

  public BridgeExample () {
  }

  /**
   * Demonstrates the use of JtBridge. It encodes
   * an object using two XML implementations.
   */

  public static void main(String[] args) {

    JtFactory factory = new JtFactory ();
    JtMessenger messenger = new JtMessenger ();
    JtBridge bridge;
    JtObject implementor;
    JtMessage msg;
    HelloWorld hello = new HelloWorld ();

    hello.setGreetingMessage("Hello World ...");

    // Create an instance of JtBridge

    bridge = (JtBridge) factory.createObject (JtBridge.JtCLASS_NAME);

    // Specify the implementor to be executed.

    implementor = (JtObject) factory.createObject (JtXMLHelper.JtCLASS_NAME);
    bridge.setImplementor(implementor);

    // Encode the object using the XML implementor selected

    msg = new JtMessage (JtObject.JtXML_ENCODE);
    msg.setMsgContent(hello);
    System.out.println ("Java encoding:" +  messenger.sendMessage (bridge, msg));

    // Use a different encoding implementation (the one provided by JtFactory)

    bridge.setImplementor(new JtFactory ());

    System.out.println ("Jt encoding:" + messenger.sendMessage (bridge, msg));



  }}
```

4.1.23. State

Jt.JtState implements the state Design pattern. The following example is extracted from Jt.examples.patterns.Switch. The concreteState attribute defines the current state.

```
/**
 * Demonstrates the use of the State design pattern.
 * The Switch class handles two possible states (OnSwitch and OffSwitch).
 */

public static void main(String[] args) {

     JtFactory factory = new JtFactory ();
     JtMessenger messenger = new JtMessenger ();
     Switch mySwitch;


     // Create an instance of JtState (Switch)

     mySwitch = (Switch) factory.createObject (Switch.JtCLASS_NAME);

     // Set the concrete state (On Switch)

     mySwitch.setConcreteState(new OnSwitch ());


     System.out.println ("switch value: " + mySwitch.getSwitchValue());

     // Flip the switch

     messenger.sendMessage(mySwitch, new JtMessage (Switch.FLIP_SWITCH));

     System.out.println ("flipping the switch: " +
                         mySwitch.getSwitchValue());

     messenger.sendMessage(mySwitch, new JtMessage (Switch.FLIP_SWITCH));

     System.out.println ("flipping the switch: " +
                         mySwitch.getSwitchValue());



}
```

4.1.24. GoF Examples

1) Calculator.java – example of Command and Factory Method.
2) Calculator1.java – example of Command (request log and undo support), Factory Method and Memento.
3) Calculator3.java – example of Command, Strategy, Memento and Factory Method. This example will be discussed in detail later on.
4) TimerComponent.java – example of Animated/Live Component (traditional multithreading). The timer component executes in an independent/separate thread.
5) Multiplication.java – example of Strategy and Factory Method.
6) Mediator.java – example of Mediator and Animated/Live Component (traditional multithreading).
7) Visitor.java – example of Visitor.
8) Flyweight.java – example of Flyweight, Abstract Factory and State.
9) DecoratedHelloWorld.java – Decorator.
10) Alarm.java – Observer and Animated/Live Component (traditional multithreading).
11) StrategyExample.java – Strategy.
12) BrigeExample.java – Bridge.
13) Switch.java – State.
14) TemplateMethodExample.java – Template Method.
15) BuilderExample.java – Builder.
16) EnglishHelloWorld.java and FrenchHelloWorld.java – Chain of responsibility.
17) TestFacade.java – Façade
18) SinglentonHelloWorld.java – Singleton

The class that implements the design pattern often includes a main method that illustrates its use (src/Jt).

Item 3 above demonstrates the use of several framework patterns including Command, Memento, Strategy and Factory Method. These patterns were used to implement a very simple calculator. Our calculator is able to process two operations (addition and multiplication). The multiplication operation is implemented using Strategy. Two possible multiplication strategies are implemented (direct multiplication or repetitive addition). The calculator can also undo the last operation by using Memento. Operations are logged and the request log is displayed before exiting. The following are the main classes involved:

JtFactory – creates instances of the other classes involved.

Calculator3 – subclass of JtCommand. It is able to process addition and multiplication operations.

JtKeyboard – reads keyboard input.

Multiplication – subclass of JtStrategy. Performs multiplications based on one of two strategies (MultiplicationA or MultiplicationB) .

MultiplicationA (concrete strategy) - standard multiplication.

MultiplicationB (concrete strategy) - performs multiplication by using repetitive addition.

JtMemento – stores the state of an object for later use (undo operation).

Calculator3 is able to handle two types of request: ADD and MULTIPLY

4.2. Jt Data Access Objects (DAO)

The Jt framework provides a native implementation of the DAO design pattern via an adapter (Jt.DAO.JtDAOAdapter). This adapter implements the DAO functionality (insert, update, delete, find, etc.) based on the JDBC adapter. It supports composite keys and transactions. The framework supports several DAO strategies including Hibernate. Additional DAO strategies can be easily plugged in. In some instances, Hibernate (heavy DAO implementation) may not be used due to resource constraints. The following section of code would Insert, Find, Update and Delete a Data Access Object:

```
String emailId = "freedom@domain.com"; // Key

 adapter = (JtDAOStrategy) main.createObject(JtDAOStrategy.JtCLASS_NAME);


 // Member object

 mem = new Member ();
 mem.setEmail(emailId);
 mem.setFirstname("John");
 mem.setLastname("Dow");
 mem.setTstamp (new Date ());
 mem.setStatus(1);


 // Insert the object to the database

 msg = new JtMessage (JtDAOAdapter.JtCREATE);
 msg.setMsgContent(mem);
 messenger.sendMessage (adapter, msg);

 // Retrieve the object from the database

 msg.setMsgId (JtDAOAdapter.JtREAD);
 msg.setMsgContent(new Member ());
 msg.setMsgData (emailId);
 mem = (Member) messenger.sendMessage (adapter, msg);

   // Update the object

 if (mem != null)
     mem.setFirstname("Jane");
 msg = new JtMessage (JtDAOAdapter.JtUPDATE);
 msg.setMsgContent(mem);
 messenger.sendMessage (adapter, msg);

 // Delete the object

 msg = new JtMessage (JtDAOAdapter.JtDELETE);
 msg.setMsgContent(mem);
 messenger.sendMessage (adapter, msg);
```

The section of code above is taken from Jt.examples.JtDAOExample.

Queries can be executed using the JtEXECUTE_QUERY message:

```
        msg.setMsgId (JtDAOAdapter.JtEXECUTE_QUERY);
        msg.setMsgContent("select * from roster");
        msg.setMsgData(new Member ());
       members = (List) messenger.sendMessage (adapter, msg);
```

All the DAO strategies use the same messages. The native Jt DAO implementation is the default. The Hibernate implementation of this example is basically the same. The only difference being that the Hibernate DAO strategy is chosen. For details, please refer to the Hibernate integration chapter.

JtDAOStrategy is based on the Strategy design pattern. It extends JtStrategy. The attribute concreteStrategyClassName defines the name of the class to be used as the DAO strategy. In order to use Hibernate, the attribute concreteStrategyClassName needs to be set as follows:

```
Jt.DAO.JtDAOStrategy.concreteStrategyClassName:Jt.hibernate.JtHibernateAdapter
```

If this attribute is not set, the default strategy (Jt.DAO.JtDAOAdapter) will be used. By changing this single attribute, the Hibernate examples from the Hibernate integration chapter can be easily configured to use the native Jt DAO strategy. Additional strategies can be plugged in by creating a new strategy class and setting the concretStrategyClassName attribute appropriately.

The additional attributes required by JtDAOdapter can be added to the framework resource file:

```
! JtDAOAdapter

Jt.DAO.JtDAOAdapter.user:root
Jt.DAO.JtDAOAdapter.password:123456
Jt.DAO.JtDAOAdapter.driver:com.mysql.jdbc.Driver
Jt.DAO.JtDAOAdapter.url:jdbc:mysql://localhost/test
```

For optimal performance,   use the datasource attribute instead of the attributes above:

```
! JtDAOAdapter

Jt.DAO.JtDAOAdapter.datasource:datasrc
```

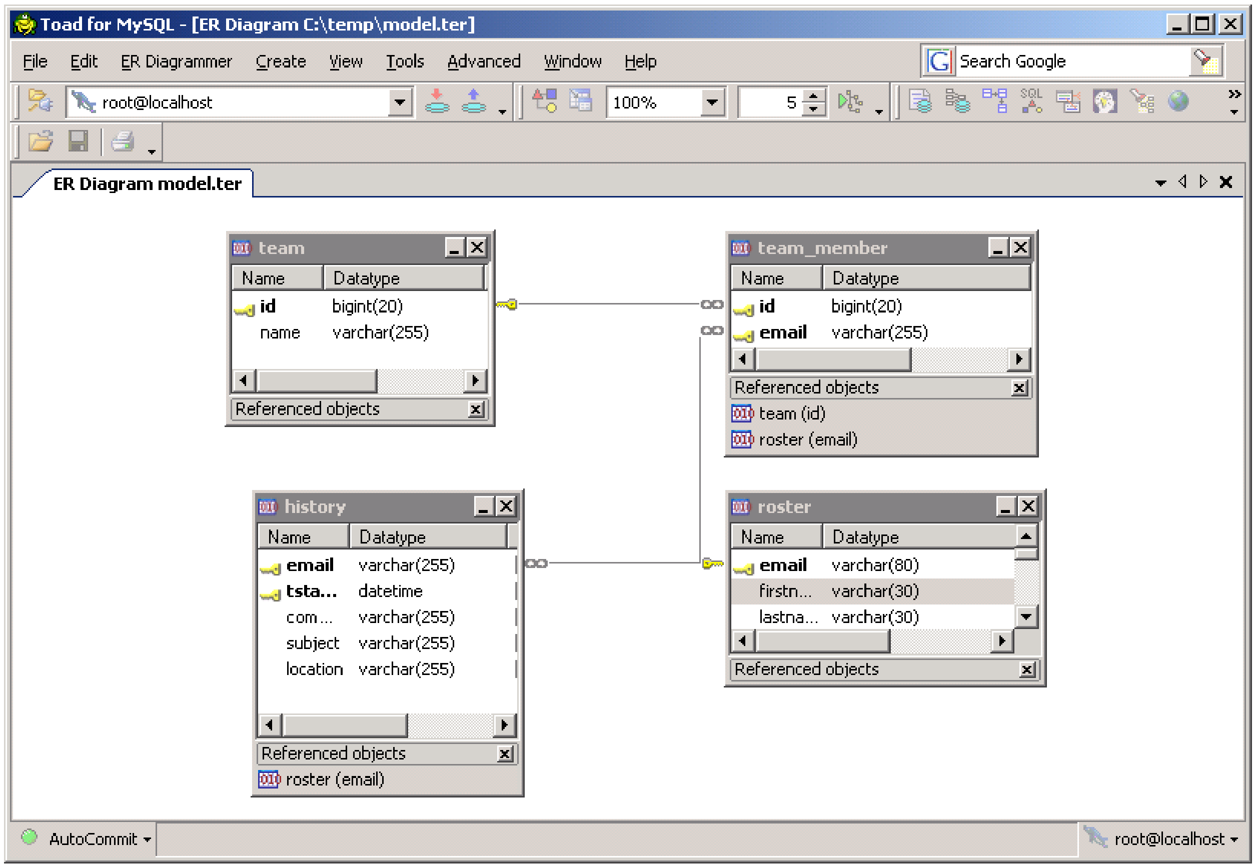
User, password, driver and url can also be added to the XML configuration file as explained in the next section. If these attributes are found in both files, the attribute values found in the configuration file are used. Keep in mind that these values are loaded from the framework resource files immediately after the JtDAOAdapter instance is created. The configuration files are read at a later time. Therefore the attribute values can be overwritten if they exist in the configuration file.

4.2.1. XML Mapping and Configuration files

The following are the MySQL Database schema and the mapping/configuration files used by our example applications. These files are the same configuration files used by the DAO Hibernate adapter. The Jt native DAO implementation is able to interpret Hibernate mapping/configuration files.Only a subset of the Hibernate syntax can be interpreted. The rest is ignored by the native DAO implementation. The Hibernate example directory (under Jt.examples.hibernate) contains several configuration/mapping files. This includes the one that illustrates a many-to-many relationship. As described before, the Hibernate examples can be configured to use the native Jt DAO strategy. The Hibernate web application can also be configured to use the native DAO implementation. Please refer to the DAO Hibernate integration section for additional details. In order to run the sample applications, you will probably need to change the database connection settings depending on your specific database configuration.

```
<?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>

    <class name="Jt.examples.hibernate.Member" table="roster">
        <id name="email">
        </id>
        <property name="firstname"/>
        <property name="lastname"/>
        <property name="comments"/>
        <property name="email_flag"/>
        <property name="subject"/>
        <property name="status"/>
        <property name="location"/>
        <property name="tstamp"/>
        <property name="mdate"/>
    </class>


</hibernate-mapping>

?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->

        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/test</property>
        <property name="connection.username">root</property>
        <property name="connection.password">123456</property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <!--<property name="hbm2ddl.auto">create</property> -->

        <mapping resource="Jt/examples/hibernate/Member.hbm.xml"/>
        <mapping resource="Jt/examples/hibernate/History.hbm.xml"/>
        <mapping resource="Jt/examples/hibernate/Team.hbm.xml"/>
        <mapping resource="Jt/examples/hibernate/TeamMemberRelationship.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

These configuration files define the database/object mappings. They also define the connection driver, url, usename and password.


Toad for MySQL - [ER Diagram C:\temp\model.ter]
File Edit ER Diagrammer Create View Tools Advanced Window Help
Search Google
root@localhost
100%
5
ER Diagram model.ter
AutoCommit
root@localhost

team

Name	Datatype
id	bigint(20)
name	varchar(255)

Referenced objects

team_member

Name	Datatype
id	bigint(20)
email	varchar(255)

Referenced objects
team (id)
roster (email)

history

Name	Datatype
email	varchar(255)
tsta...	datetime
com...	varchar(255)
subject	varchar(255)
location	varchar(255)

Referenced objects
roster (email)

roster

Name	Datatype
email	varchar(80)
firstn...	varchar(30)
lastna...	varchar(30)

Referenced objects

```
create table history (email varchar(255) not null, tstamp datetime not null,
comments varchar(255), subject varchar(255), location varchar(255), primary
key (email, tstamp));
create table roster (email varchar(255) not null, firstname varchar(255),
primary key (email));
alter  table  history  add  index  FK620B70742619AF3  (email),  add  constraint
FK620B70742619AF3 foreign key (email) references roster (email);

create table team (id bigint not null auto_increment, name varchar(255),
primary key (id));
create table team_member (id bigint not null, email varchar(255) not null,
primary key (id, email));
alter table team_member add index FKA29B52BC2619AF3 (email), add constraint
FKA29B52BC2619AF3 foreign key (email) references roster (email);
alter  table  team_member  add  index  FKA29B52BC6E071F75  (id),  add  constraint
FKA29B52BC6E071F75 foreign key (id) references team (id);
```

4.2.2. Transactions

The native Jt DAO strategy (JtDAOAdapter) handles transactions automatically. This is the default behavior. Transactions can also be handled manually. The transaction is started by sending `JtDAOAdapter.JtBEGIN_TRANSACTION` to the adapter. After this, the transaction can be manually committed or rolled back. Jt.examples.HBDAOExample1 and Jt.examples.HBDAOExample3 illustrate the use of manual transactions:

```
        // Create the Hibernate adapter

        adapter = (JtDAOStrategy) factory.createObject (JtDAOStrategy.JtCLASS_NAME);

        // Start a transaction

       messenger.sendMessage (adapter, new JtMessage (JtDAOAdapter.JtBEGIN_TRANSACTION));

        //… Additional operations

        …

        // Commit the changes (JtROLLBACK rolls back the changes)

        messenger.sendMessage (adapter, new JtMessage (JtDAOAdapter.JtCOMMIT));
```

4.3. JavaMail Adapter

The Jt Framework provides an adapter for the JavaMail API. The documentation for JtMail can be found under Jt/dist/docs/api. Several JtPortal components make use of the JavaMail Adapter. For instance Jt.portal.Register and Jt.wizard.MailingList.

```
  /**
   * Demonstrates the use of the framework Java Mail adapter.
   */

    public static void main (String[] args) {
        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();
        JtMail jtmail;


        // Create an instance of JtMail

        jtmail = (JtMail) factory.createObject (JtMail.JtCLASS_NAME);


        // Set JtMail attributes

        jtmail.setFrom("Jt@yahoo.com");
        jtmail.setSubject("Hello World");
        jtmail.setMessage("Welcome to Jt messaging!");
        jtmail.setTo("email@yahoo.com");


        // Activate JtMail (send email)

        messenger.sendMessage (jtmail, new JtMessage (JtObject.JtACTIVATE));


        // Check for exceptions (it any)

        if (jtmail.getObjException() == null)
            System.out.println  ("JtMail: GO");
        else
            System.out.println  ("JtMail: FAIL");


    }
  }
```

The additional attribute values required by the JtMail component can be added to the framework resource file:

```
! Java Mail adapter

Jt.JtMail.server:myserver.com
Jt.JtMail.username:user
Jt.JtMail.password:password
Jt.JtMail.port:587
```

4.4. JDBC Adapter

The Jt Framework provides an adapter for the JDBC API. The documentation for JtJDBCAdapter can be found under Jt/dist/docs/api. This JDBC adapter supports transactions, data sources and transparent BPM integration. It is also the basis for the Jt implementation of the DAO design pattern (Jt.DAO.*). The following piece of code extracted from Jt.JDBCAdapter illustrates the use of JtJDBCAdapter.

```
    /**
     * Demonstrates the use of the JtJDBCAdapter.
     */

    private static void test () {
        JtFactory factory = new JtFactory ();  // JtFactory
        JtMessage msg = new JtMessage (JtJDBCAdapter.JtQUERY);
        JtMessenger messenger = new JtMessenger ();
        JtJDBCAdapter adapter;

        // Create an instance of JtJDBCAdapter

        adapter = (JtJDBCAdapter) factory.createObject (JtJDBCAdapter.JtCLASS_NAME);

        // Execute a query

        msg.setMsgContent ("select email from roster where email='member@hotmail.com';");
        messenger.sendMessage (adapter, msg);

        // Illustrates the use of transactions. Begin a transaction

        messenger.sendMessage (adapter, new JtMessage (JtJDBCAdapter.JtBEGIN_TRANSACTION));

        // Execute the query

        msg = new JtMessage (JtJDBCAdapter.JtEXECUTE_UPDATE);
        msg.setMsgContent("update roster set email_flag=1;");
        messenger.sendMessage (adapter, msg);

        // Commit the transaction

        messenger.sendMessage (adapter, new JtMessage (JtJDBCAdapter.JtCOMMIT));

        // Warning: Jt version 7.4 (and above) requires that JtCLOSE be sent
        // manually. JtREMOVE is not longer automatically sent via removeObject.

        messenger.sendMessage (adapter, new JtMessage (JtJDBCAdapter.JtCLOSE));
        factory.removeObject (adapter);

    }
```

The attributes required by JtJDBCAdapter can be added to the framework resource file:

```
! JDBC adapter

Jt.JtJDBCAdapter.user:root
Jt.JtJDBCAdapter.password:123456
Jt.JtJDBCAdapter.driver:com.mysql.jdbc.Driver
Jt.JtJDBCAdapter.url:jdbc:mysql://localhost/test

! For better performance use the datasource attribute instead of the 4 above
! Jt.JtJDBCAdapter.datasource:datasrc
```

4.5. JMS Adapters

The Jt Framework provides integration with the JMS API. Two JMS adapters are included: JtJMSQueueAdapter (point-to-point) and JtJMSTopicAdapter (publish/subscribe). The project page contains Javadoc documentation for these adapters. Program examples that demonstrate the use of the JMS and ESB adapters can be found under src/java/Jt/examples/jms and jtwizard/src/java/Jt/rftp. Very few simple lines of code are needed to add JMS functionality to Jt framework applications. For instance, the following section of code would send a Jt message to a JMS queue:

```
// Create the JMS adapter (point-to-point)


jmsAdapter = (JtJMSQueueAdapter)
   factory.createObject (JtJMSQueueAdapter.JtCLASS_NAME);

// Send a message to the JMS queue via the JMS adapter.


messenger.sendMessage (jmsAdapter, message); // message is an instance of any
                                             // serializable class


// Receive a message from the JMS queue via the JMS adapter


messenger.sendMessage(jmsAdapter, new JtMessage (JtJMSAdapter.JtRECEIVE));
```

The JtRftp application relies on the JMS adapter. This reference application can be downloaded from the project page. In order to run the JMS examples, JMS needs to be configured appropriately. Jt.properties contains settings for several attributes. These settings need to match your JMS configuration. For example:

```
Jt.jms.JtJMSQueueAdapter.queue:testQueue

Jt.jms.JtJMSQueueAdapter.connectionFactory:TestJMSConnectionFactory

Jt.jms.JtJMSQueueAdapter.timeout:1
```

4.6. JNDI Adapter

JNDI integration is performed via the JtJNDIAdapter. In order to use this adapter, several object attributes need to be set: url, user, password and factory.

```
    public static void main (String[] args) {
        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();
        JtJNDIAdapter adapter;
        Object entry;
        JtMessage msg = new JtMessage (JtJNDIAdapter.JtLOOKUP);


        // Create an instance of JtJNDIAdapter

        adapter = (JtJNDIAdapter)
             factory.createObject (JtJNDIAdapter.JtCLASS_NAME);

        // Send a JtLOOKUP message to the adapter

        msg.setMsgContent ("JtSessionFacade");

        entry = (Object) messenger.sendMessage (adapter, msg);

        if (entry != null)
            System.out.println  ("JtJNDIAdapter: GO");
        else
            System.out.println  ("JtJNDIAdapter: FAIL");

        // Remove the JtJNDIAdapter instance

        factory.removeObject (adapter);


    }
```

As usual these attributes can be included in Jt.properties. For instance:

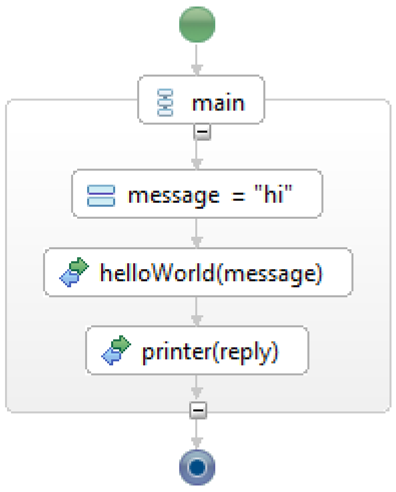
```
! JNDI Adapter

Jt.jndi.JtJNDIAdapter.factory:weblogic.jndi.WLInitialContextFactory
Jt.jndi.JtJNDIAdapter.url:t3://localhost:7001
```

5. BPEL integration and MDP extensions

The Jt framework features integration with the business process execution language (BPEL). A BPEL engine based on MDP is provided within the Jt framework. This engine implements the BPEL extensions that are required to support the MDP messaging approach. These extensions are aimed to provide support for components (local and remote), services and messaging. The handling of local components, remote components and multithreading applications is simplified and improved by using the MDP model.

The BPEL software development process is centered around a business process. A business process is modeled using a process graph. This minimizes the need for coding and increases productivity and quality. It also provides users with a very powerful way of modeling business processes. The Jt API documentation contains a detailed description of the framework classes used in support of BPEL. Sample applications can be found under the processes directory.

The following process graph represents the Hello World application:



Simple BPEL diagram of HelloWorld. A message (‘hi’) is sent to the helloWorld component. The reply is sent to the printer component which prints it. The message sent to the MDP component is included inside parenthesis. This application/graph can be created using the Eclipse BPEL designer or any other comparable BPEL tool.

As shown above, MDP components can be readily included in BPEL diagrams. This is one of the powerful advantages of MDP.

The following is the BPEL/XML representation of the HelloWorld process graph. It includes references to the component mentioned above.

```
<!-- Hello World MDP/BPEL Process [Generated by the Eclipse BPEL Designer]  -->
<!-- Features the Messaging Design Pattern (MDP) extensions -->
<!-- Date: Tue Apr 06 17:09:55 EDT 2010 -->

<bpel:process name="HelloWorld"
         targetNamespace="http://eclipse.org/bpel/sample"
         suppressJoinFailure="yes"
         xmlns:tns="http://eclipse.org/bpel/sample"
         xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
         abstractProcessProfile="http://docs.oasis-
open.org/wsbpel/2.0/process/abstract/simple-template/2006/08"
xmlns:ns1="http://www.w3.org/2001/XMLSchema">


    <bpel:partnerLinks>

    </bpel:partnerLinks>

    <!-- ================================================================= -->
    <!-- VARIABLES                                                         -->
    <!-- List of components used within this BPEL process                  -->
    <!-- ================================================================= -->
    <bpel:variables>
          <bpel:variable name="helloWorld"
              type="java:Jt.examples.HelloWorldMessage"></bpel:variable>
          <bpel:variable name="printer" type="java:Jt.JtPrinter"></bpel:variable>
          <bpel:variable name="reply" type="java:java.lang.String"></bpel:variable>
          <bpel:variable name="message" type="java:java.lang.String"></bpel:variable>
    </bpel:variables>

    <!-- ================================================================= -->
    <!-- ORCHESTRATION LOGIC                                               -->
    <!-- Set of activities coordinating the flow of messages across the    -->
    <!-- components integrated within this business process                -->
    <!-- ================================================================= -->

    <bpel:sequence name="main">

       <!-- Assign the String "hi" to message -->
        <bpel:assign validate="no" name='message  = "hi"'>
             <bpel:copy>
                <bpel:from>"hi"</bpel:from>
                <bpel:to>$message</bpel:to>
            </bpel:copy>
        </bpel:assign>

        <!-- Send the message (hi) to the helloWorld component -->
        <bpel:invoke name="helloWorld(message)" component="helloWorld"
            inputVariable="message" outputVariable="reply" >
        </bpel:invoke>

        <!-- Send the reply to the printer component -->
        <bpel:invoke name="printer(reply)"
            component="printer" inputVariable="reply"></bpel:invoke>

    </bpel:sequence>
</bpel:process>
```

5.1. MDP process components

This section describes the BPEL extensions required to support the MDP component model. Variables can be declared using the object class associated to the variable. Variables that are not explicitly initialized are implicitly initialized to point to an instance of the class. The initialization is performed when the variable is first used. In other words, the user does not need to initialize the variable. An assumption is made here: if the variable is declared, it is safe to assume that it should be initialized with a value different than null (object instance). A future version should support explicit initialization while the variable is being declared.

In the above example, the instance of HelloWorldMessage is initialized during the execution of the Invoke activity. The initialization is performed only once.

```
<bpel:variable name="helloWorld"
   type="java:Jt.examples.HelloWorldMessage"></bpel:variable>
<bpel:variable name="printer" type="java:Jt.JtPrinter"></bpel:variable>
<bpel:variable name="reply" type="java:java.lang.String"></bpel:variable>
<bpel:variable name="message" type="java:java.lang.String"></bpel:variable>
<bpel:variable name="I" type="java:java.lang.Integer"></bpel:variable>
<bpel:variable name="D" type="java:java.lang.Double"></bpel:variable>
<bpel:variable name="C" type="java:java.lang.Character"></bpel:variable>
<bpel:variable name="B" type="java:java.lang.Boolean"></bpel:variable>
<bpel:variable name="exception" type="java:java.lang.Exception">
```

Component attributes can be accessed by using the following syntax:

$variable.attributeName

For instance:

$helloWorld.greetingMessage

Many BPEL activities require the use of expressions (arithmetic, boolean, relational, logical, etc):

$I + 1*3

($D/2 + 1.0) < 5.0

$helloWorld.greetingMessage==”hi”

$helloWorld.greetingMessage==null

$helloWorld.greetingMessage!=null && helloWorld.greetingMessage==”hi”

5.2. BPEL Activities

The MDP messaging interface is implemented via the BPEL invoke activity. A few attributes have been added to this activity:

Component (required): name of the variable that references the recipient component.

Exception (optional): name of the variable that stores the exception object (if any). This variable is used to store the exception detected while the recipient component is processing the input message.

Asynchronous (optional): attribute that specifies whether or not the message should be sent asynchronously. The default is false (synchronous processing). The invoke activity does not use the standard way of extending BPEL (ExtensionActivity). The current implementation was designed to fit the Eclipse BPEL designer GUI. The invoke activity may need to be changed in future releases.

<Invoke> Activity

```
<bpel:invoke name="helloWorld(message)" component="BPELVariableName"
       inputVariable="BPELVariableName" outputVariable="BPELVariableName"
         exception="BPELVariableName"
       asynchronous="true|false">
          activity
       <bpel:catch faultMessageType="QName">*
          activity
       </bpel:catch>
       <bpel:catchAll>?
          activity
       </bpel:catchAll>
</bpel:invoke>
```

Let’s take a look at some examples:

```
<!-- Send the message (hi) to the helloWorld component -->
<bpel:invoke name="helloWorld(message)" component="helloWorld"
    inputVariable="message" outputVariable="reply" >
</bpel:invoke>
```

This example uses inputVariable (input message) and outputVariable (reply).

The following invoke activity handles exceptions. faultMessageType specifies the exception type (object class):

```
<bpel:invoke name="helloWorld(message)" component="helloWorld"
       inputVariable="message" outputVariable="reply" exception="exception">


       <bpel:catch faultMessageType="java:java.lang.Exception">
              <bpel:sequence>
                     <bpel:invoke name="printer(message)" component="printer"
                            inputVariable="exception"></bpel:invoke>

              </bpel:sequence>
       </bpel:catch>
       <bpel:catchAll>

              <bpel:sequence>


                     <bpel:invoke name="printer(exception)" component="printer"
                            inputVariable="exception"></bpel:invoke>

              </bpel:sequence>
       </bpel:catchAll>

</bpel:invoke>
```

The semantic is very similar to the one used by the Java try-catch block. If an exception is detected while processing the message, it is handled by the <catch> block associated with the exception type. In other words, the exception type must match the block's faultMessageType. If the exception doesn't match any of the types specified by the catch blocks, it is handled by the <catchAll> block.

The following BPEL activities are supported:

<sequence> Activity

```
<bpel:sequence>
      activity+
</bpel:sequence>
```

<if> Activity

```
<bpel:if >
      <bpel:condition>boolean expression</bpel:condition>
          activity
      <bpel:elseif>*
            <bpel:condition>boolean expression</bpel:condition>
            activity
      </bpel:elseif>
      <bpel:else>?
            activity
      </bpel:else>
</bpel:if>
```

<while> Activity

```
 <bpel:while>
      <bpel:condition> boolean expression</bpel:condition>
             activity
 </bpel:while>
```

<repeatUntil> Activity

```
  <bpel:repeatUntil
      <bpel:condition>boolean expression</bpel:condition>
             activity
  </bpel:repeatUntil>
```

<empty> Activity

```
<bpel:empty></bpel:empty>
```

<exit> Activity

```
<bpel:exit></bpel:exit>
```

<throw> Activity

```
<bpel:throw faultVariable="BPELVariableName"></bpel:throw>
```

BPELVariableName is the name of the variable that stores the exception object.

<rethrow> Activity

```
        <bpel:rethrow></bpel:rethrow>
```

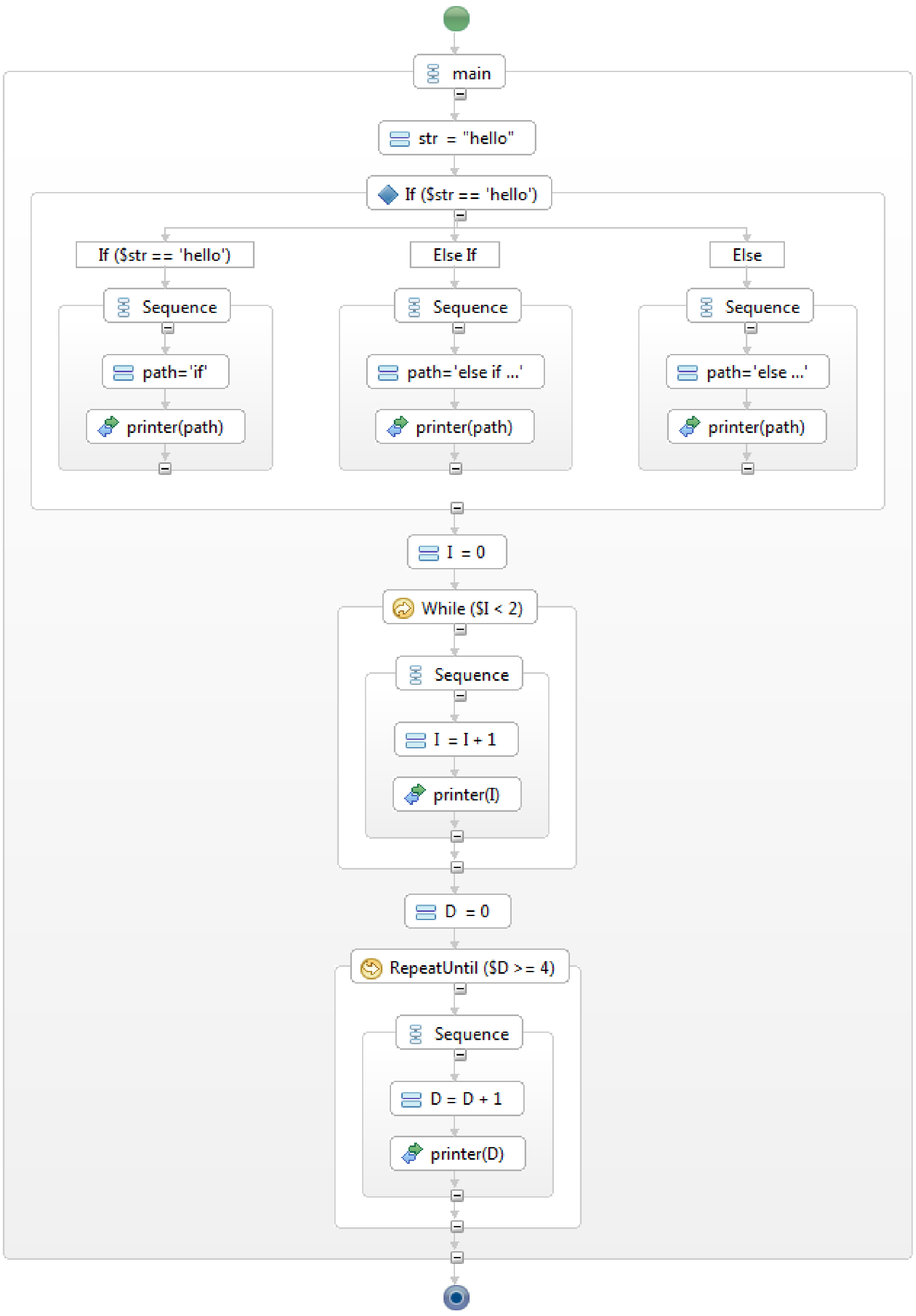
This activity can only be used with <Catch> and <CatchAll>

<Assign> Activity

```
        <bpel:assign>
             <bpel:copy>
                <bpel:from>expression</bpel:from>
                <bpel:to>$variableName</bpel:to>
            </bpel:copy>
        </bpel:assign>
```

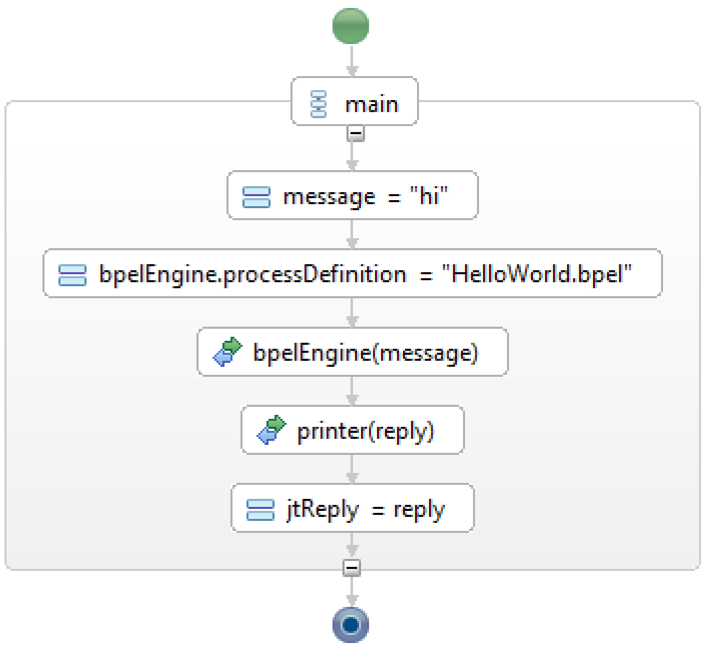
The MDP model does not need to implement all the BPEL activities and features. Partner Links, for instance, are implemented using the MDP capabilities which provide transparent access to remote components and services via MDP Proxies: remote components are treated as local objects. MDP provides a simplified model.

The <flow> activity is another activity that is not required by the MDP model. The conventional way of implementing concurrency (multithreading) does not seem to follow a natural pattern. In the real world, the concept of a concurrent thread is inherently coupled to a component (see Animated or Live Component pattern). Both concepts don't come separately. We find that certain components are able to act independently from the rest (concurrency). We can view each one of these components as having their own thread. Concurrency is implemented via the synchronous attribute of the invoke activity: if the synchronous attribute is false, the message is processed asynchronously by the component's independent thread.


main
str = "hello"
If ($str == 'hello')
If ($str == 'hello')
Else If
Else
Sequence
path='if'
printer(path)
Sequence
path='else if ...'
printer(path)
Sequence
path='else ...'
printer(path)
I = 0
While ($I < 2)
Sequence
I = I + 1
printer(I)
D = 0
RepeatUntil ($D >= 4)
Sequence
D = D + 1
printer(D)

5.3. Process Composition

A process can contain one or more sub processes. This allows the user to build a complex processes by splitting it into more manageable sub processes. Jt.bpel.JtBPELEngine is the MDP component responsible for executing BPEL processes. The process definition attribute needs to be set. The execution of a process may return a reply via the variable specified by outputVariable (Invoke Activity). The process needs to set the reply variable ($jtReply) before exiting.



The following sections of code illustrate how a subprocess should be invoked:

```
<bpel:assign validate="no" name='bpelEngine.processDefinition  = "HelloWorld.bpel"'>
     <bpel:copy>
        <bpel:from>"/home/projects/Jt/HelloWorld.bpel"</bpel:from>
        <bpel:to>$bpelEngine.processDefinition</bpel:to>
    </bpel:copy>

</bpel:assign>

<!-- Send the message (hi) to the subprocess -->
<bpel:invoke name="bpelEngine(message)" component="bpelEngine"
    inputVariable="message" outputVariable="reply"></bpel:invoke>

<!-- Update the jtReply variable -->
<bpel:assign validate="no" name='jtReply  = reply'>
     <bpel:copy>
        <bpel:from>$reply</bpel:from>
        <bpel:to>$jtReply</bpel:to>
    </bpel:copy>
</bpel:assign>
```

5.4. Invoking and Debugging the BPEL example

Type one of the following commands to invoke the BPEL example:

```
java Jt.bpel.JtBPELEngine HelloWorld.bpel
```

or

```
java –DLog=stderr Jt.jbpel.JtBPELEngine HelloWorld.bpel
```

A hello message should be displayed. The second command will have logging/debugging information sent to the screen. You can invoke these commands directly from the Eclipse IDE. JtBPELEngine takes one parameter: the name of the process definition file to be executed.

6. MDP Components

The following are some of the main components provided by the Jt Framework out of the box. The Jt API documentation includes a complete list of framework components (JtFile, JtDirectory, JtURL, JtInputStream, JtKeyboard, JtOSCommand, JtPrinter, JtEcho, JtLogger, JtRegitry, JtExceptionHandler, JtMessageDigest, JtXMLHelper, etc). These reusable components can be easily plugged into new applications. The framework also provides a set of reusable components for security and encrypted messaging. The Jt reference applications (JtPortal and JtWizard) contain a large number of reusable components (see the Java source). Some components should be mainly used via BPEL/BPM/UML diagrams or remote access (JtCollection, JtIterator and JtList).

6.1. **MDP and traditional multithreaded applications**

Framework components that inherit from JtAnimatedComponent implement the Animated or Live Object pattern. This design pattern deals with many of the complexities associated with traditional multithreaded applications. A paper describing this design pattern is being submitted for publication.

6.2. Live or Animated Object Pattern

Intent: This design pattern encapsulates component functionality, processing (threading) mechanism and the messaging functionality required to provide the component with independent behavior (a “live of its own” so to speak). This also means that the component uses its own independent processing mechanism or thread of execution. This design pattern improves decoupling, encapsulation and scalability while at the same time reducing complexity and overall implementation cost. Component functionality, processing/treading mechanism and messaging mechanism are decoupled entities, independent of one another. MDP messaging allows the interchange of information (i.e. messages) between the animated component and other components or applications. Although decoupled and independent of one another, processing/threading mechanism and component functionality are completely encapsulated within a single entity: live or animated object.

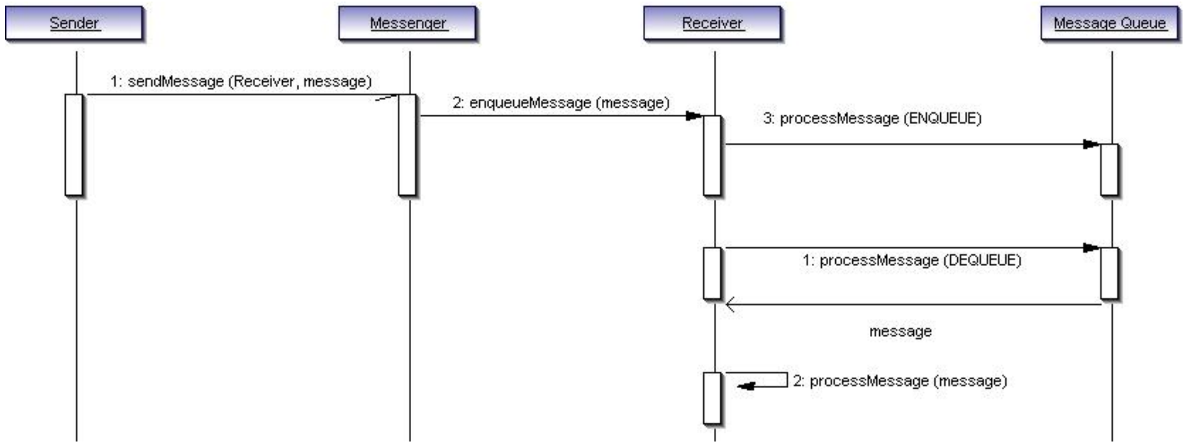
Applicability: This design pattern can be applied in wide variety of scenarios. Live or animated objects are all around us in the real world. Living beings, computers, automated systems, machines and components of these entities represent a few examples. All these entities exhibit an independent behavior and can be modeled as animated or live objects. Messaging is the mechanism of communication between animated components and their external environment including other components or applications. This approach is not only realistic; it also handles many of the complexities associated with the implementation of traditional multithreaded applications.

Motivation: The implementation of traditional multithreaded application is a complex undertaking which usually becomes costly, time consuming and prone to error. Defects related to are often encountered (thread management, synchronization, race conditions, deadlocks, etc). These software defects are difficult to avoid, reproduce and isolate. Large multithreaded applications complicate the problem even further. The degree of complexity and risk considerably worsens as the number of threads and their interactions increases. Object oriented applications consist of a collection of components that interact with each other. In reality, some of these components should be modeled as live or animated components: They exhibit independent behavior, a “life of their own”. For instance, a banking application may need to model a teller component able to process multiple banking transactions concurrently. This component runs using its own thread. There will probably be several teller components running independently within the context of a banking application. Notice that the teller component should encapsulate the component functionality and the processing mechanism as two independent concerns within the component. On the other hand, they are both part of a single component (teller) and should not be artificially modeled or implemented as separate objects. In general, the implementation of an animated component calls for one single thread which provides the processing mechanism. Messaging provides the mechanism of communication. Another real world example would be a highly interactive smartphone application able to concurrently speak, recognize voice, play music, manage the graphical user interface and while concurrently performing other operations (calls, text messages, notifications, alarms, etc). No single thread is allowed to monopolize cpu utilization for long periods of time, otherwise the offending thread may be force to stop. A realistic and accurate model for this application would consist of several animated components communicating with each other via messaging. This solution avoids and solves many of the issues associated with a comparable multithreaded application including complexity, coupling, lack of encapsulation, reusability, scalability, and overall implementation cost.

Participants:

a) Live or Animated object: Component that encapsulates component functionality and independent processing/threading mechanism which provides the component with its independent behavior. Live or animated objects communicate with other components via messaging as defined by the messaging design pattern (MDP): components, message and messaging mechanism are decoupled entities, fully independent.

b) Message Queue: internal subcomponent used to enqueue the incoming messages. The component relies on a queuing mechanism to store the incoming messages being sent asynchronously. The component's thread or processing mechanism dequeues the messages and processes them one at a time. A MDP framework usually implements the logic required to manage the component thread and the message queue. The MDP component only needs to inherit this functionality from a framework superclass.


Sender
Messenger
Receiver
Message Queue
1: sendMessage (Receiver, message)
2: enqueueMessage (message)
3: processMessage (ENQUEUE)
1: processMessage (DEQUEUE)
message
2: processMessage (message)

MDP Asynchronous messaging

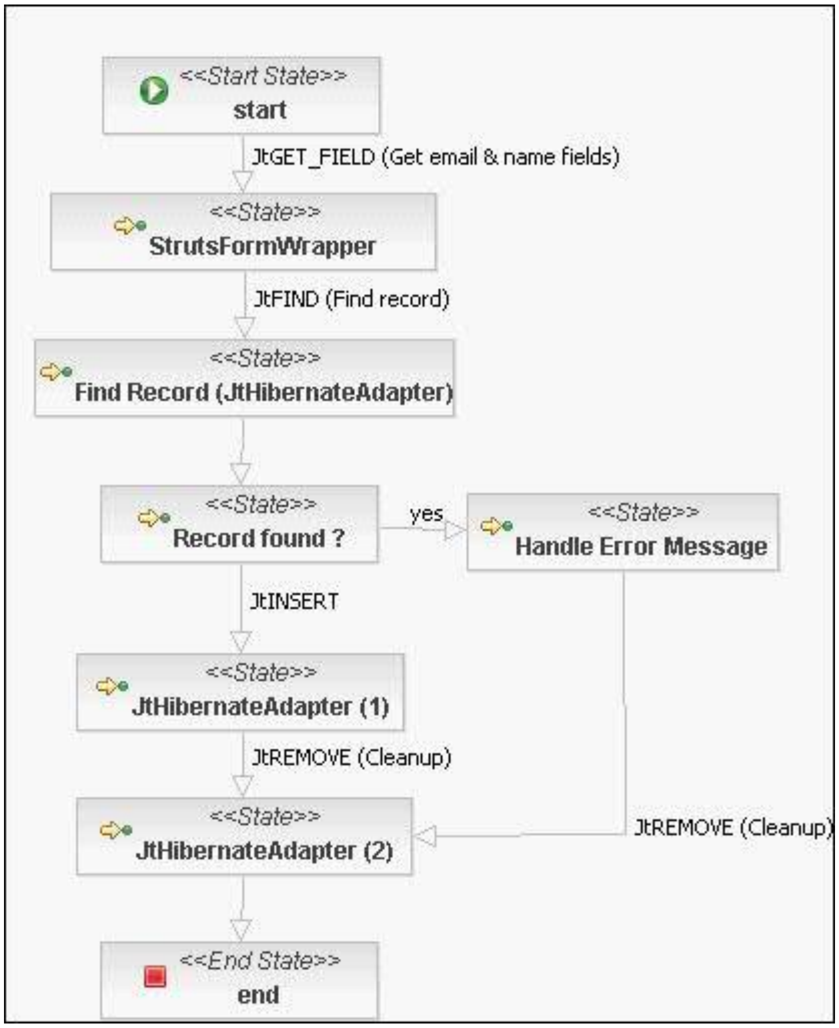
The following example is taken from Jt.examples.patterns.TimerComponent:

```
/**
 * MDP implementation of a timer. The timer implements the Live or
 * Animated Component Pattern and runs using its own independent
 * thread of execution. Most of the asynchronous
 * messaging and threading functionality is implemented by the MDP
 * framework.
 */

public class TimerComponent extends JtAnimatedComponent {
    private static final long serialVersionUID = 1L;
    public static final String JtCLASS_NAME = TimerComponent.class.getName();
    public static final String UPDATE_TIME = "UPDATE_TIME";
    private long tstart = 0L;        // t0
    private long tend;               // t1
    private double time;             // Elapsed time in seconds (delta)
    private JtFactory factory = new JtFactory ();
    private JtLogger logger = null;  // Logger component

    public TimerComponent() {
       logger = (JtLogger) factory.lookupObject(JtFactory.jtLogger);
    }

    // Attributes

    public double getTime () {
        return (time);
    }
    public void setTime (double time) {
        this.time = time;
    }

    private void updateTime () {

        if (tstart == 0L)
                    tstart = (new Date()).getTime ();

              tend = (new Date ()).getTime ();
              time = (tend - tstart)/1000.0;

    }

    // Process component messages

    public Object processMessage (Object message) {

        Object msgid = null;
        JtMDPMessage msg = (JtMDPMessage) message;

        if (msg == null)
            return null;

        msgid = factory.getValue(msg, JtMDPMessage.ID);

        if (msgid == null)
            return null;
        // Start/Update the timer

        if (msgid.equals (TimerComponent.UPDATE_TIME) ||
                    msgid.equals (JtComponent.JtACTIVATE)) {
```

```
             updateTime ();

            // Another UPDATE_TIME request is added to the message queue
             this.enqueueMessage(new JtMDPMessage (TimerComponent.UPDATE_TIME));

            return (null);
        }
        // Let the superclass handle JtSTOP (stop the thread of execution)

        if (msgid.equals (JtComponent.JtSTOP))
                    return (super.processMessage (message));


        logger.handleError ("Invalid message:" + msgid);
        return (null);

    }
    static private char waitForInputKey () {
        char c = ' ';

        try {

            c = (char) System.in.read ();
            while (System.in.available () > 0)
                System.in.read ();

        } catch (Exception e) {
            e.printStackTrace ();
        }

        return (c);
    }
    // Demo program
    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();
        TimerComponent timer;
        // Create the Timer component

        timer = (TimerComponent) factory.createObject (TimerComponent.JtCLASS_NAME);

        System.out.println ("Press any key to start the timer ....");
        waitForInputKey ();
        System.out.println ("Timer started ....
        // Start the timer component (animated component).
        // The MDP framework is responsible for implementing
        // the processing or threading mechanism. This message is
        // processed asynchronously via a message queue.

        messenger.setSynchronous(false);
        messenger.sendMessage (timer, new JtMDPMessage (JtComponent.JtACTIVATE));
        System.out.println ("Press any key to stop the timer ....");
        waitForInputKey ();
        System.out.println (timer.getTime() + " second(s) elapsed.");
        // Stop the timer
        messenger.sendMessage (timer, new JtMDPMessage (JtComponent.JtSTOP));
    }
}
```

7. Hibernate DAO integration

The Jt framework provides a native implementation of the DAO design pattern. Jt also provides and adapter that supports the Hibernate DAO implementation (Jt.hibernate.JtHibernateAdapter). The adapter implements the DAO functionality (insert, update, remove, find, etc.). The following BPM diagram represents a demo application that uses the Jt Hibernate adapter.



This BPM process is part of a mailing list demo application that requests email address and name from the user via a web/struts form. The information provided by the user is stored using JtHibernateAdapter (JtINSERT message). If the entry already exists, an error message is produced. This application also uses the MVC desing pattern (JtStrutsAdapter).

Sample programs that demonstrate the use of the Hibernate adapter can be found under src/java/Jt/examples/hibernate. A detailed description of the JtHibernateAdapter messaging API can be found in the Jt API documentation under the docs/api directory.

Very few simple lines of code are needed to integrate Hibernate with Jt framework applications. For instance, the following section of code would Insert, Find, Update and Delete a Data Access Object:

```
 JtMessage msg;
 JtHibernateAdapter adapter;
 String emailId = "user@freedom.com"; // Key
 Member mem;
 List Members;


// Create the Hibernate adapter

adapter = (JtHibernateAdapter) factory.createObject (JtHibernateAdapter.JtCLASS_NAME);


 // Member object

 mem = new Member ();
 mem.setEmail(emailId);
 mem.setFirstname("John");
 mem.setLastname("Dow");
 mem.setTstamp (new Date ());
 mem.setStatus(1);


 // Insert object

 msg = new JtMessage (JtDAOAdapter.JtCREATE);
 msg.setMsgContent(mem);
 messenger.sendMessage (adapter, msg);

// Find (Read) an Object

 msg = new JtMessage (JtDAOAdapter.JtREAD);
 msg.setMsgContent(mem);
 msg.setMsgData (emailId);  // Key (identifier)
 mem = (Member) messenger.sendMessage (adapter, msg);


 // Update the object

 if (mem != null)
    mem.setFirstname("Jane");

 msg = new JtMessage (JtDAOAdapter.JtREAD);
 msg.setMsgContent(mem);
messenger.sendMessage (adapter, msg);

 // Delete the object

 msg = new JtMessage (JtDAOAdapter.JtDELETE);
 msg.setMsgContent(mem);
 msg.setMsgData(emailId);
 messenger.sendMessage (adapter, msg);
```

Queries can be executed using the JtEXECUTE_QUERY message:

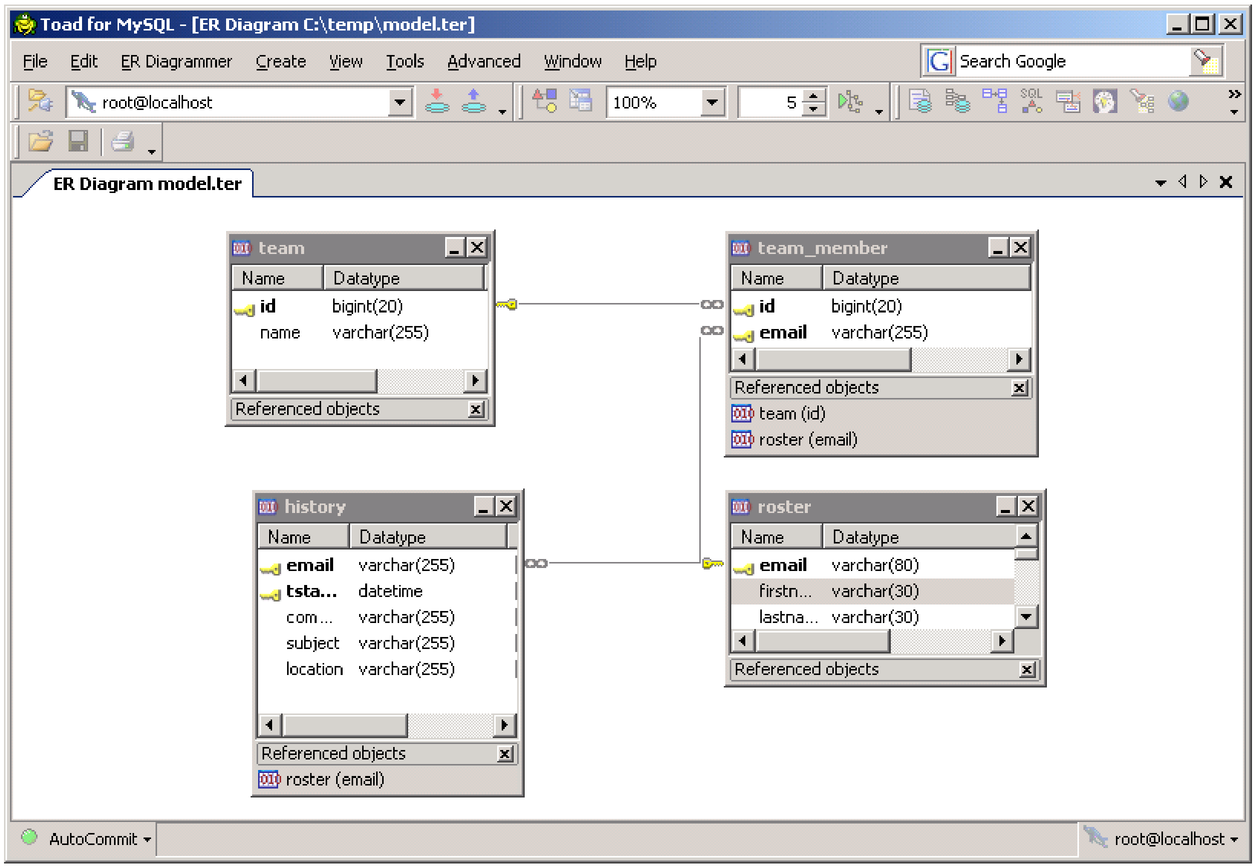
```
 msg.setMsgId (JtDAOAdapter.JtEXECUTE_QUERY);
 msg.setMsgContent("select * from roster");
 msg.setMsgData(new Member ());
 members = (List) messenger.sendMessage (adapter, msg);
```

The Hibernate DAO adapter (JtHibernateAdapter) handles transactions automatically. This is the default behavior. JtHibernateAdapter commits the transaction after each operation. If an exception is detected, JtHibernateAdapter rolls back the transaction. Transactions can also be handled manually. The transaction is started by sending `JtDAOAdapter.JtBEGIN_TRANSACTION` to the adapter. After this, the transaction can be manually comitted or rolled back. Jt.examples.HBDAOExample1 illustrates the use of manual transactions:

```
        // Create the Hibernate adapter

        adapter = (JtHibernateAdapter) factory.createObject (JtHibernateAdapter.JtCLASS_NAME);


        // Start a transaction

       messenger.sendMessage (adapter, new JtMessage (JtDAOAdapter.JtBEGIN_TRANSACTION));


        //… Additional operations


        // Commit the changes (JtROLLBACK rolls back the changes)

        messenger.sendMessage (adapter, new JtMessage (JtDAOAdapter.JtCOMMIT));
```

The following are the MySQL Database schema and the Hibernate mapping and configuration files used by our demo applications. You will probably need to change the database dialect and connection settings depending on your specific database configuration.

```
<!-- Member.hbm.xml -->
<?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>

    <class name="Jt.examples.hibernate.Member" table="roster">
        <id name="email">
        </id>
        <property name="firstname"/>
    </class>


</hibernate-mapping>
```


Toad for MySQL - [ER Diagram C:\temp\model.ter]
File Edit ER Diagrammer Create View Tools Advanced Window Help
Search Google
root@localhost
100%
5
ER Diagram model.ter
AutoCommit
root@localhost

team

Name	Datatype
id	bigint(20)
name	varchar(255)

Referenced objects

team_member

Name	Datatype
id	bigint(20)
email	varchar(255)

Referenced objects
team (id)
roster (email)

history

Name	Datatype
email	varchar(255)
tsta...	datetime
com ...	varchar(255)
subject	varchar(255)
location	varchar(255)

Referenced objects
roster (email)

roster

Name	Datatype
email	varchar(80)
firstn...	varchar(30)
lastna...	varchar(30)

Referenced objects

```
create table history (email varchar(255) not null, tstamp datetime not null,
comments varchar(255), subject varchar(255), location varchar(255), primary
key (email, tstamp));
create table roster (email varchar(255) not null, firstname varchar(255),
primary key (email));
alter table history add index FK620B70742619AF3 (email), add constraint
FK620B70742619AF3 foreign key (email) references roster (email);

create table team (id bigint not null auto_increment, name varchar(255),
primary key (id));
create table team_member (id bigint not null, email varchar(255) not null,
primary key (id, email));
alter table team_member add index FKA29B52BC2619AF3 (email), add constraint
FKA29B52BC2619AF3 foreign key (email) references roster (email);
alter table team_member add index FKA29B52BC6E071F75 (id), add constraint
FKA29B52BC6E071F75 foreign key (id) references team (id);
```

```
<!-- hibernate.cfg.xml -->
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->

        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/test1</property>
        <property name="connection.username">root</property>
        <property name="connection.password">1234567</property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <!-- <property name="dialect">org.hibernate.dialect.HSQLDialect</property> -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>


        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <!--<property name="hbm2ddl.auto">create</property> -->

        <mapping resource="Jt/examples/hibernate/Member.hbm.xml"/>
        <mapping resource="Jt/examples/hibernate/History.hbm.xml"/>
        <mapping resource="Jt/examples/hibernate/Team.hbm.xml"/>
        <mapping resource="Jt/examples/hibernate/TeamMember.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

```
<!-- History.hbm.xml -->
<?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>

    <class name="Jt.examples.hibernate.History" table="history">
      <composite-id>
        <key-property name="email"/>
        <key-property name="tstamp" type="timestamp"/>
      </composite-id>
        <property name="comments"/>
        <property name="subject"/>
        <property name="location"/>

    </class>

</hibernate-mapping >
```

7.1. Invoking the Hibernate DAO sample applications

In order to run the sample applications, you will need to configure Hibernate to use your database. Please refer to the Hibernate documentation for complete information on how to do this. The following are minimum steps that may need to be completed depending on the database configuration:

1) Download the sample web application from http://jt.dev.java.net (JtHibernateExamples under documents & files). This application demonstrates the combined use of the Hibernate, Struts and BPM adapters. The source files can be found under webapps/JtHibernateExamples

2) Change the Hibernate configuration file (hibernate.cfg.xml).You will probably need to change the database dialect and connection settings depending on your specific database configuration. The web application contains a copy of hibernate.cfg.xml under /WEB-INF/classes.

3) Configure the JDBC database driver associated to the target database. The driver file may need to be added to /WEB-INF/lib. The MySQL driver file is already part of the war file.

4) Create the database tables listed above. This can be done manually. Hibernate can create these tables automatically by uncommenting a line in the configuration file:

```
        <!-- Drop and re-create the database schema on startup - Existing data will be lost -->
        <property name="hbm2ddl.auto">create</property>
```

5) Deploy the web application and redirect your browser to the appropriate URL. Tomcat uses http://localhost:8080/JtHibernateExamples.

In order to use Hibernate, the attribute concreteStrategyClassName has been added to the framework resource file:

```
Jt.DAO.JtDAOStrategy.concreteStrategyClassName:Jt.hibernate.JtHibernateAdapter
```

If this line is commented out, the native Jt DAO implementation will be used.

The demo applications under src/java/Jt/examples/hibernate can be executed (HBDAOExample, HBDAOExample1 and HBDAOExample2). This can be done directly from the IDE, such as Eclipse, or the command line.   These applications illustrate the use of composite keys and the association mappings. In order to run these applications, you will need to complete similar configuration steps (Hibernate and JDBC drivers). hibernate.cfg.xml can be found under src/config.files. The database tables don't need to be recreated. According to the Hibernate FAQ there is an issue with the <key-many-to-one> mapping. Therefore <key-many-to-one> could not be used to model the many-to-one relationships. For instance, the relationship between the history table and the roster table. A fix for this issue is expected in future versions of Hibernate.

7.2. Modeling many-to-many relationships

Jt.examples.hibernate.HBDAOExample2 illustrates a many-to-many relationship between teams and team members. As expected an additional table is needed to model this type of relationship. It is recommended that a separate class be used. The following Hibernate mapping file describes the relationship between teams and team members.

```
<?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>

    <class name="Jt.examples.hibernate.TeamMemberRelationship"
table="team_member">
      <composite-id>
        <key-property name="id"/>
        <key-property name="email"/>
      </composite-id>

    </class>
</hibernate-mapping>
```

Using a separate class presents several advantages. This approach is consistent with the relational model (a separate entity (table) is used). It is also a very simple approach. Finally the same JtHibernateAdapter API described above can be used here as well. For example:

```
        // Add a member to the team

        team_member = new TeamMemberRelationship ();
        team_member.setId(team.getId());
        team_member.setEmail(mem.getEmail());

        msg = new JtMessage (JtDAOAdapter.JtINSERT);
        msg.setMsgContent(team_member);
       messenger.sendMessage (adapter, msg);

        // Retrieve all the members from a team

        messenger.sendMessage (adapter, new JtMessage (JtDAOAdapter.JtBEGIN_TRANSACTION));

        msg.setMsgId (JtDAOAdapter.JtEXECUTE_QUERY);
        msg.setMsgContent("select r.email, r.firstname from team_member tm, roster r, team t
where tm.id=t.id and tm.email = r.email and t.name = 'Team Blue'");
        msg.setMsgData(new Member ());
        members = (List) messenger.sendMessage (adapter, msg);

       // Retrieve all the teams a member belongs to.

        msg.setMsgId (JtDAOAdapter.JtEXECUTE_QUERY);
        msg.setMsgContent("select t.id, t.name from team_member tm, team t where tm.id=t.id and
tm.email = 'daniel@jt.net'");
        msg.setMsgData(new Team ());
        teams = (List) messenger.sendMessage (adapter, msg);
```

8. Java Server Pages (JSP) integration

The Jt Framework provides support for several J2EE technologies including JSPs. The following simple steps will allow you to use the Jt Framework within JSPs:

A) Import the Jt Framework classes (Jt.*).

B) Use the <jsp:useBean/> tag to create the Jt factory.

C) Use JSP scriplets and expressions to implement the business logic by sending messages to the Jt business objects.

D) View the results.

Source code for the sample web applications can be found under webapps. The war file JtWebExamples.war can be downloaded from http://jt.dev.java.net (JtSampleApps.zip under documents & files). Complete JSP applications can be downloaded from the same location (JtWizard.war and JtPortal.war). The following example illustrates the steps above (Jt/webapps/JtWebExamples/HelloWorld.jsp):

```
<jsp:root version="1.2" xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:c="http://java.sun.com/jstl/core">
  <jsp:directive.page contentType="text/html; charset=ISO-8859-1" />
<jsp:directive.page import="Jt.*" />
<jsp:directive.page import="Jt.examples.*" />
<html>
<head>
<title>HelloWorld.jsp - Example of JSP support</title>
</head>
<body>

<!-- HelloWorld.jsp - Example of JSP support -->
<!-- Create the component factory  -->
<jsp:useBean id='factory' class='Jt.JtFactory' scope='request'/>
<!-- Create the helloWorld component -->

<jsp:scriptlet> HelloWorldMessage helloWorld = (HelloWorldMessage) factory.createObject
                                   (HelloWorldMessage.JtCLASS_NAME);
</jsp:scriptlet>

<!-- Implement the business logic by sending a message to the component  -->

<jsp:scriptlet> String jtReply = (String) factory.sendMessage (helloWorld, “hi”);
                request.setAttribute ("jtReply", jtReply);
</jsp:scriptlet>

<!-- View the reply message -->

<c:out value="${jtReply}"/>

</body>

</html>

</jsp:root>
```

9. Servlet Integration

Applications based on servlets can be easily integrated within the Jt Framework. A universal Jt servlet (Jt.servlet.JtServet) provides a transparent interface between the servlet API and framework components.

To use the JtServlet you will need to add the following Servlet definition to the web.xml file.

```
<servlet>
  <servlet-name>JtServlet</servlet-name>
  <servlet-class>Jt.servlet.JtServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>JtServlet</servlet-name>
  <url-pattern>/JtServlet</url-pattern>
</servlet-mapping>
```

In general every Jt component can be accessed using JtServlet, provided that access has been granted:

```
http://localhost:8080/JtPortal/JtServlet?classname=Jt.examples.HelloWorld&msgId=JtHELLO

http://localhost:8080/JtPortal/JtServlet?classname=Jt.examples.DateService&msgId=DISPLAY_DATE
```

For security reasons, no access is granted unless you explicitly enable it by configuring the framework access manager:

Jt.security.JtAccessManager.classAccessList:Jt.examples.*,Jt.portal.service.*

This grants remote access to the packages Jt.examples and Jt.portal.sevice. The access manager (JtAccessManager) is one of the framework components responsible for granting remote access to framework objects. A later section explains the framework access manager in detailed. JtServlet is the basis for the framework integration with REST and AJAX.

As mentioned earlier, the resource file (Jt.properties) needs to be in the classpath. For web applications, Jt.properties is usually placed under *WEB-INF/classes.*

9.1. Messaging Information

As shown in the previous section, Jt messaging information needs to be passed to JtServlet. This component expects the following parameters:

a) msgId: Jt message Id (optional. If msgId is not present, JtACTIVATE is passed)

b) msgContent: Jt message content (optional)

c) msgData: Jt message data (optional)

d) msgAttachment: Jt message attachment (optional)

These parameters are optional. They may be used to pass additional messaging data. JtServlet also expects an aditional parameter, className: name of the remote component that will process the message. The remote class needs to be a Java bean and implement JtInterface. If you prefer not to pass the className parameter, you may want to consider the Struts integration instead. No className is required when invoking framework Struts. You can also define subclasses of JtServlet and hard code the the remote class. In this case, you will need one servlet per each class.

These parameters are consistent with the Jt messaging API. The reply object resulting from the execution of the delegate class is sent back to the client. Reply messages (Java objects of type other than java.lang.String) need to be converted to its XML representation before being returned to the client. This is automatically handled by JtServet. Java objects of type String are not converted. The Jt framework contains several helper classes that convert Java objects into XML and vice versa. These classes can be found under the package Jt.xml. For additional details, please refer to the section dealing with the XML integration. In our specific HelloWorld example, the reply is a String object (“Hello World …”). This string is returned to the client (no need for XML conversion).

The following invocation returns a Date instance:

```
http://localhost:8080/JtPortal/JtServlet?classname=Jt.examples.DateService&msgId=DISPLAY_DATE
```

In this case, the reply is converted to its XML representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_06" class="java.beans.XMLDecoder">
 <object class="java.util.Date">
  <long>1257731352722</long>
 </object>
</java>
```

10. MDP Framework Access Manager

The access manager (Jt.server.JtAccessManager) is the framework component responsible for granting remote access to framework components. Framework applications that use ESB, EJBs, JtServlet, REST, AJAX and Web Services rely on the access manager. For security reasons, no access is granted unless you explicitly enable it by configuring the framework access manager. This can be done by updating the framework properties file:

Jt.security.JtAccessManager.classAccessList:Jt.examples.HelloWorld

This grants remote access to the class Jt.examples.HelloWorld.

Jt.security.JtAccessManager.classAccessList:Jt.examples.HelloWorld,Jt.portal.service.*

This line, on the other hand, grants remote access to the class Jt.examples.HelloWorld and all the classes in the Jt.portal.service package.

As shown above, the user can grant access to a collection of classes and packages. Attempts to access other framework components via the remote interfaces (JtServlet, AJAX, REST, EJBs and Web Services) will generate a security exception. Access will be denied.

The framework also provides user/role based security. It relies on the standard capabilities provided by HttpSession. The HttpSession instance is used to store the JtContext. The framework remote interfaces take advantage of these capabilities. The JtPortal application shows how this is accomplished. For instance, the component Jt.serviceLogging can be invoked after the user (admininistrator role) logs in. This REST service enables/disables logging. Users that don’t belong to the administrator role will be denied access.

The framework access manager can also be configured via the JtAccessManager.xml resource file. This file needs to be part of the CLASSPATH. If JtAcessManager.xml is present, the framework properties file is ignored. The following example grants access to the package Jt.examples.*, the class Jt.examples.EchoService, the class Jt.service.Logging and the component jtEchoComponent.

```
<?xml version="1.0" encoding="UTF-8"?>
<Object classname="Jt.security.JtAccessManager">



<classEntryAccessList>
  <Object classname="java.util.ArrayList">
    <Object classname="Jt.security.JtClassAccessEntry">
      <enabled>True</enabled>
      <classname>Jt.examples.*</classname>
      <encrypted>false</encrypted>
    </Object>
    <Object classname="Jt.security.JtClassAccessEntry">
      <enabled>true</enabled>
      <classname>Jt.examples.EchoService</classname>
      <encrypted>true</encrypted>
    </Object>
    <Object classname="Jt.security.JtClassAccessEntry">
      <enabled>true</enabled>
      <classname>Jt.service.Logging</classname>
      <encrypted>true</encrypted>
      <authenticated>true</authenticated>
    </Object>

  </Object>

</classEntryAccessList>

<componentEntryAccessList>
  <Object classname="java.util.ArrayList">
    <Object classname="Jt.security.JtComponentAccessEntry">
      <enabled>true</enabled>
      <entryId>jtEchoComponent</entryId>
      <encrypted>true</encrypted>
    </Object>
  </Object>
</componentEntryAccessList>


</Object>
```

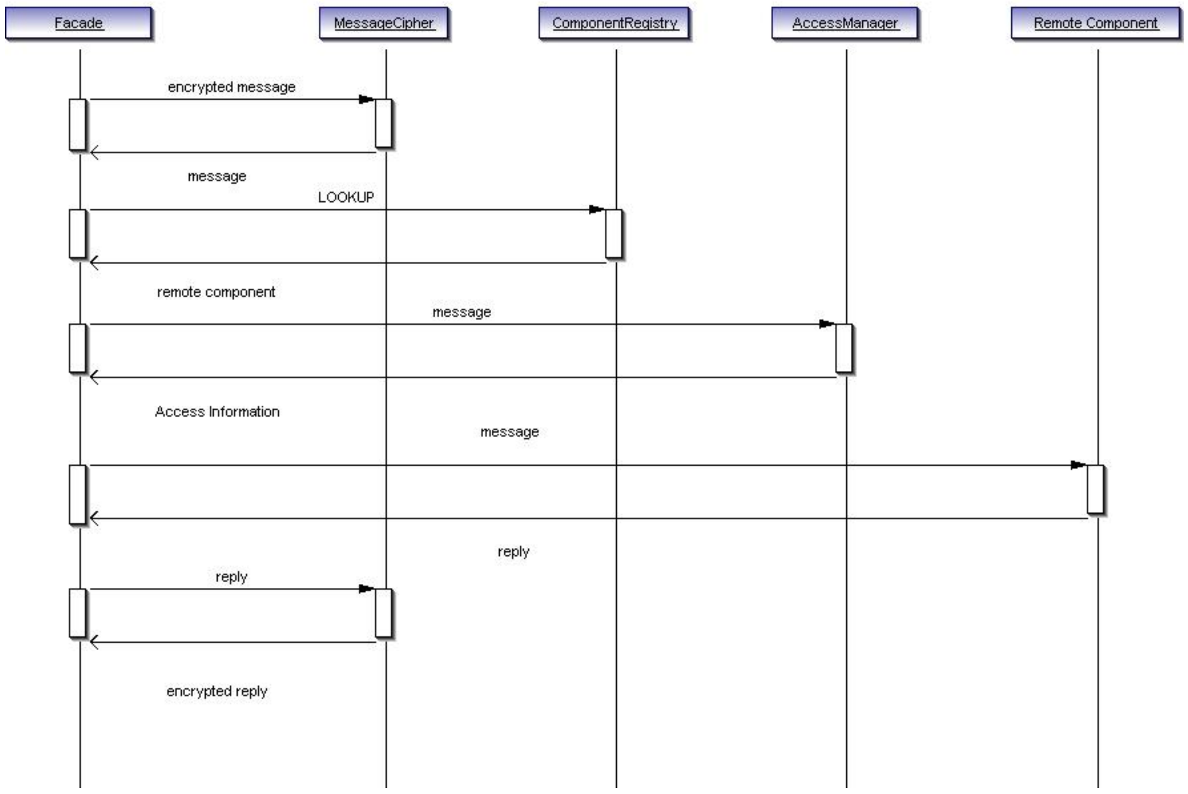
Access to Jt.examples.EchoService and jtEchoComponent must be performed via encrypted/secure messaging: the framework automatically encrypts each message before it is sent to the remote component. The framework also encrypts the reply message automatically. For additional information, please refer to the section dealing with framework security and encryption. Encrypted/secure messaging is supported by the framework web services (Axis & REST).

Access to Jt.service.Logging is encrypted and authenticated. Messages sent to this class need to contain valid username and password (JtContext). The authentication is performed via JAAS and the JtJAASAdapter. The user is able to specify the component that verifies the username/password pair. For additional information, please refer to the framework security section.

11. MDP Framework Security and Secure Messaging

The Jt messaging design pattern can be employed to provide strong security and encryption mechanisms:

```
public interface JtInterface  {

/**
  * Jt messaging interface used for the implementation
  * of the messaging design pattern.
  * Process a message and return a reply.
  */

  Object processMessage (Object message);


}
```



The following are the components involved:

MessageCipher: component responsible for decrypting the input message and encrypting the reply message. This component can be configured to use a specific encryption scheme.

Component Registry: allows the system to register and look up components by ID.

AccessManager: responsible for granting/denying access to remote components. It authorizes and authenticates each message received. If the access manager is unable to authenticate the message, it never reaches the receiver.

Well-known encryption and authentication mechanisms fit in well with the messaging design pattern. Since framework messages can be instances of any class, the user can provide custom encryption and authentication mechanisms to implement security. The sender can encrypt the message before sending it. In turn, the recipient can decrypt/authenticate the message before processing it. On the other hand, the framework modules dealing with Web Services (Axis/Rest) already implement encryption and security capabilities. The following reusable components are provided:

a) Jt.security.JtMessageCipher: encrypts the messages interchanged between components and/or applications. It uses several well-know encryption mechanisms: session key encryption, symmetric encryption and asymmetric encryption. Messages are encrypted using the symmetric session key provided by the sender. If no session key is provided, the framework generates one. The session key is then encrypted with the public key of the recipient.
b) Jt.security.JtAsymmetricCipher: this component implements asymmetric encryption.
c) Jt.security.JtSymmetricCipher: this component implements symmetric encryption.
d) Jt.security.JtEncryptedMessage: Encrypted messages. It includes the encrypted message and the encrypted session key.
e) Jt.security.KeyStore: Keystore component.
f) Jt.security.Certificate: Certificates
g) Jt.security.JtMessageAuthenticator: authenticates messages. It relies on the JAAS capabilities provided via JtJAASAdapter.
h) Jt.JAAS.JtJAASLoginModule: JAAS Login Module used for message authentication.

These components can be configured to use a specific encryption/authentication mechanism, key size, certificate, keystore, etc. The Jt properties file contains the following default values:

```
! Jt KeyStore (client application)

Jt.security.JtKeyStore.password:password
Jt.security.JtKeyStore.alias:Jt

Jt.security.JtKeyStore.resourcePath:JtClient.ks



! Jt KeyStore (server application)

Jt.security.JtKeyStore.password:password

Jt.security.JtKeyStore.alias:Jt

Jt.security.JtKeyStore.resourcePath:JtServer.ks

! Jt Asymmetric Cipher

Jt.security.JtAsymmetricCipher.certificateName:Jt

Jt.security.JtAsymmetricCipher.transformation:RSA/ECB/PKCS1Padding

! Jt Symmetric Cipher

Jt.security.JtSymmetricCipher.algorithm:Blowfish

Jt.security.JtSymmetricCipher.transformation:Blowfish/ECB/PKCS5Padding

Jt.security.JtSymmetricCipher.keySize:128
```

To generate keystore and certificate, run keytool as usual. For instance, the following command generates JtServer.ks using RSA:

```
keytool -genkey -alias Jt -keyalg RSA -keystore JtServer.ks
```

The certificate needs to be exported and imported into the client’s truststore:

```
keytool -export -alias Jt -file Jt.cer -keystore JtServer.ks

keytool -import -keystore JtClient.ks -alias Jt -file Jt.cer
```

The keystore file needs to be part of the server’s CLASSPATH. The truststore file needs to be part of the client’s CLASSPATH. In summary, you will need to do the following in order to enable secure/encrypted messaging between local and remote components:

a) Create or update the keystore (server application). Export the server's certificate. The keystore file needs to be part of the server's CLASSPATH. Web applications, for instance, usually include the keystore under *WEB-INF/classes*.
b) Update the server's properties file (Jt.properties) using the appropriate values for the KeyStore class (resourcePath, alias, password).
c) Update the access manager configuration file (server). This file should allow access to the remote class/component. It should also specified encrypted messaging (encrypted attribute).
d) Create or update the truststore (client application). The server's certificate needs to be imported into the client's truststore.
e) Update the client's properties file (Jt.properties) using the appropriate values for the KeyStore class (resourcePath, alias, password).
f) The service invocation (client application) should specify encrypted messaging. Please refer to the sections dealing with web services (Axis proxy and REST services).
g) Authenticated services require username and password to be passed as part of the message (JtContext). Please refer to the sections dealing with web services.

Message authentication is provided by JAAS and JtJAASAdapter. JAAS needs to be configured. The following is the JAAS configuration file (jaas.config):

```
Jt {

  Jt.JAAS.JtLoginModule required

  loginVerificationClassname="Jt.examples.LoginStrategy";

};
```

loginVerificationClassname specifies the component that performs the username/password validation. This component contains two attributes (username and password). Jt.examples.LoginStrategy is an example that illustrates how this component needs to be implemented.

java.security may need to reference the JAAS configuration file shown above:

```
#
# Default login configuration file
#

login.config.url.1=file:/home/projects/Jt/jaas.config
```

12. Model View Controller (MVC) Design Pattern

Apache Struts implements the MVC (Model View Controller) design pattern. Struts applications can be easily integrated within the Jt Framework. The business logic (controller piece) can be implemented using Jt framework components and/or BPM business processes. A universal Jt action (Jt.struts.JtStrutsAction) provides a transparent interface between the Jt framework API and Struts. This universal action also helps reduce the number of Struts actions that are required by the application.

12.1. JtStrutsAction

The Jt framework contains several applications that illustrate the use of Struts within the Jt framework. JtWizard and JtPortal make heavy use of the Struts integration. Let's take a look at the HelloWorld action under Jt/webapps/JtStrutsExamples. To deploy the universal Jt action, add the following entry to the actions mappings section of the Struts configuration file:

```
<action     path="/HelloWorld"
            type="Jt.struts.JtStrutsAction"
            parameter="Jt.examples.HelloWorld"
            scope="session">
            <forward name="success"             path="/HelloWorld.jsp"/>
            <forward name="failure"             path="/index.jsp"/>
</action>
```

The parameter specifies the delegate class that will implement the business logic. JtStrutsAction invokes the delegate class. This class has to implement the JtInterface. It should also be a Java bean containing the attributes objException and objErrors. objException is used to store any exception detected while processing messages. objErrors stores the list of error messages that should be displayed by the View component (usually a JSP). In our example application, Jt.example.HelloWorld will be the delegate class. The HelloWorld class was covered in detail in previous chapters. The action parameter can also be used to specify a BPEL process definition file:

```
<action  path="/BPMHelloWorld"
      type="Jt.struts.JtStrutsAction"
      parameter="/WEB-INF/processes/hello.bpel"
            scope="session">
      <forward name="success"             path="/HelloWorld.jsp"/>
      <forward name="failure"             path="/index.jsp"/>
</action>
```

In the scenario above, the business logic is implemented via a BPEL process.

12.2. The View Component

The reply resulting from the execution of the delegate class or the BPM process is placed in the HttpServletRequest by the JtStrutsAction. jtReply is the attribute used to store the reply:

```
            // Set the jtReply attribute

            request.setAttribute("jtReply", jtReply);
```

This piece of code is part of JtStrutsAction. jtReply is made available to the View component. In our case, jtReply is made available to HelloWorld.jsp. Keep in mind that since the Jt framework is a messaging system, every component returns a reply after processing each input message.

```
<c:out value="${jtReply}"/>


or

<%=request.getAttribute ("jtReply") %>
```

This JSP code displays the reply resulting from the execution of the Struts action. The following url should be used to invoke the actions above:

<html:link page="/HelloWorld.do?msgId=JtHELLO">Hello World action</html:link>

or

<html:link page="/BPMHelloWorld.do">BPEL Hello World action</html:link>

Notice that the JtHello message Id is included as a parameter when a delegate class is used. This message Id is sent to the HelloWorld class. The BPEL action doesn’t require a message Id.

12.3. Jt Messaging information

As shown in the Struts application example, Jt messaging information can be passed to the delegate class or the BPEL process:

Action.do?msgId=JtHELLO&msgContent=somecontent&msgData=somedata

A Jt message is sent to the delegate class:

a) msgId: Jt message Id (optional. If msgId is not present, JtACTIVATE is passed when a delegate class is used).

b) msgContent: Jt message content (optional).

c) msgData: Jt message data (optional).

d) msgAttachment: Jt message data (optional).

e) msgAsynchronous: Use asynchronous messaging (optional:true or false)

12.4. Jt Context

JtStrutsAction also passes the Jt Context to the delegate class or BPM process. The JtContext contains ActionForm and HttpServlet information (context, request, response, etc). This information may be required by the delegate class for the implementation of custom and advanced features. JtContext is part of the Jt package.

```
public class JtContext implements Serializable {

    public static final String JtCLASS_NAME = JtContext.class.getName();
    private static final long serialVersionUID = 1L;
    private Object servletContext;          // Servlet context
    private Object actionForm;              // Struts action form
    private HttpServletRequest request;     // Servlet request
    private HttpServletResponse response;   // Servlet reponse

…


}
```

12.5. ActionForm

This section illustrates ActionForm information being passed. The code is also part of the example application Jt/webapps/JtStrutsExamples.). The form includes a single field (email). The war file (JtStrutsExamples.war) can be downloaded from http://jt.dev.java.net (JtSampleApps.zip under documents & files). A complete reference application based on Struts and JtStrutsAction can be downloaded from the same location (JtWizard.war).

```
<!-- ========== Form Bean Definitions ===================================->

<form-beans>
  <form-bean name="dynamicLookupForm"
    type="org.apache.struts.action.DynaActionForm">
    <form-property name="email"
      type="java.lang.String"/>

    </form-bean>

</form-beans>
```

The delegate class is Jt.examples.struts.FindRecords. The action performs a seach based on the email contained in the form.

```
  <action    path="/FindRecords"
             type="Jt.struts.JtStrutsAction"
             name="dynamicLookupForm"
             parameter="Jt.examples.struts.FindRecords"
             scope="session">
             <forward name="success"             path="/results.jsp"/>
             <forward name="failure"             path="/index.jsp"/>
  </action>
```

Jt.examples.struts.FindRecords.java illustrates how the JtMessaging information is passed to the delegate class. The following pieces of code are part of FindRecords.java:

```
      public Object processMessage (Object message) {
          Object data;
          ActionForm form = null;
          JtContext context;

          JtMessage e = (JtMessage) message;

          if (e == null ||  (e.getMsgId() == null))
              return (null);


          if (e.getMsgId().equals(JtObject.JtACTIVATE)) {

              // pass the form information
              context = (JtContext) e.getMsgContext();
              if (context != null)
```

```
                form = (ActionForm) context.getActionForm();

            return ( findRecords ((DynaActionForm) form));
        }

        handleError ("invalid message id:"+ e.getMsgId());
        return (null);
    }

    public Object findRecords (DynaActionForm form) {

        if (form == null) {
            return (null);
        }

        // Retrieve the email field from the ActionForm
        email = (String) form.get ("email");

     }
```

12.6. Error and Exception management

The struts ActionErrors collection will be updated if errors/exceptions are detected during the execution of the delegate class or the BPEL business process. This is automatically handled by the JtStrutsAction. The delegate class doesn't need to do anything special besides updating the objException and objErrors attributes appropriately. This is usually done by invoking handleException() or handleError(). Java beans that implement the Jt Interface will also need to declare and handle these two attributes:

private Exception objException;

private Object objErrors;

JtStrutsAction depends on having these attributes when handling exceptions and errors. Subclasses of JtObject inherit them. Please refer to the section dealing with the Jt handling of exceptions and errors. The Struts resource bundle file will need to contain the key `jt.exception:`

```
jt.exception={0}
```

As usual the <html:error /> is used by the JSP page to display the ActionErrors collection. Our sample application relies on the following resource bundle file as defined in the Struts configuration file (struts-config.xml):

```
<!-- ========== Message Resources Definitions ===========================>

<message-resources parameter="MessageResources" />
```

The following example is part of the JtWizard sample application. It represents a complete delegate class invoked by JtStrutsAction:

```
public class CreateForm extends JtObject {
    private static final long serialVersionUID = 1L;

    /*
     * Update the list of errors (objErrors) that
     * should be displayed by the View component (JSP).
     */
    private void handleUIError (String error) {
        Collection col;

        if (error == null)
            return;
        col = (Collection) this.getObjErrors();

        if (col == null) {
            col = new LinkedList ();
            setObjErrors (col);
        }
        col.add(error);

    }

    public Object generateView (DynaActionForm form) {

        String className;
        String pack;
        String formName;
        FormGenerator formGenerator = new FormGenerator ();
        Boolean dynaActionForm;

        Object jtReply;
        if (form == null) {
            return (null);
        }
        pack = (String) form.get ("pack");

        if (pack != null && !pack.equals (""))
            formGenerator.setPack(pack);

        className = (String) form.get ("className");

        if (className == null || className.equals ("")) {
            handleUIError ("Please enter a class name.");
        }
        formGenerator.setClassName(className);

        formName = (String) form.get ("formName");
        if (formName == null || formName.equals ("")) {
            handleUIError ("Please enter a form name.");
        }
        formGenerator.setFormName(formName);

        if (getObjErrors() != null) {
            return (null);
        }
        dynaActionForm = (Boolean) form.get ("dynaActionForm");
        if (dynaActionForm != null) {
            handleTrace ("dynaActionForm:" + dynaActionForm.booleanValue());
            formGenerator.setDynaActionForm(dynaActionForm.booleanValue());
        }

        jtReply = formGenerator.processMessage(new JtMessage (JtObject.JtACTIVATE));

        // Propagate the exception
        if (formGenerator.getObjException() != null)
            this.setObjException(formGenerator.getObjException());

        return (jtReply);
    }

    public Object processMessage (Object message) {
        Object data;
        JtMessage e = (JtMessage) message;
        ActionForm form = null;
        JtContext context;
        if (e == null ||  (e.getMsgId() == null))
            return (null);
        if (e.getMsgId().equals(JtObject.JtACTIVATE)) {

            // pass the form information

            context = (JtContext) e.getMsgContext();
            if (context != null)
               form = (ActionForm) context.getActionForm();
            return ( generateView ((DynaActionForm) form));
        }

        return (super.processMessage(message));

    }
}
```

13. AJAX Integration

AJAX can be easily integrated within the Jt framework. Several built-in JavaScript/AJAX and Jt components are provided:

- JtFactory and JtMessage (*scripts/Jt.js*) - Core JavaScript classes responsible for implementing the Jt messaging design pattern.

- JtAjaxAdapter (*scripts/Jt.js*) - JavaScript class responsible for sending and processing the AJAX request.

- Jt.struts.JtStrutsAjaxAction - This component allows you to invoke Struts actions via AJAX. It is a subclass of the JtStrutsAction, the framework universal Jt action.

All these components implement the Jt messaging design pattern in order to send the AJAX request. The user only needs to provide components to implement the business logic. This includes a callback to process the AJAX message reply. The rest is taken care by the framework, including the conversion of the reply message. The reply message can be a basic String or any other Java class. If it is not a String, the object is automatically converted into its XML representation using the capabilities provided by the framework. For additional information about the XML conversion, please refer to the XML integration section.

The JavaScript JtAjaxAdapter class is responsible for submiting the AJAX request. JtAjaxAdapter has several attributes:

i) url: url that is passed to the server. This is usually the url associated to the Struts action.

j) FormName: Name of the HTML form. The form values may be passed as part of the AJAX request. JtAjaxAdapter automatically extracts the form values.

k) callback: Callback provided by the user to handle the AJAX message reply.

l) operation: “GET” or “POST” (default). JtAjaxAdapter is able to handle either one.

Sections of the JtAjaxAdapter class are shown on the next pages.

```
// Jt Ajax Adapter. Submits the AJAX request.

function JtAjaxAdapter () {

 // Attributes

 this.url = null;         // url that should serve the AJAX request
 this.formName = null;    // a HTML form may be passed as part of the AJAX request
 this.callback = null;    // callback provided by the user to process the reply
 this.operation = "POST"; // POST or GET request
 this.classname = null;   // delegate Java class

 // Request object

 if (window.XMLHttpRequest) {
   this.request = new XMLHttpRequest ();

 }
 else if (window.ActiveXObject) {
   this.request = new ActiveXObject ("Microsoft.XMLHTTP");
 }

 // Invoke Ajax request

 this.invokeAjaxRequest = function (msg) {
 var params;
 var getUrl;
 params = 'msgId=' + msg.msgId;

 // Messaging information

 if (msg.msgContent != null) {
    params += '&' + 'msgContent=' + msg.msgContent;
 }

 if (msg.msgData != null) {
    params += '&' + 'msgData=' + msg.msgData;
 }

 if (msg.msgAttachment != null) {
    params += '&' + 'msgAttachment=' + msg.msgAttachment;
 }

 if (this.classname != null) {
    params += '&' + 'jtClassName=' + this.classname;
 }

 if (this.formName != null) {
    params += '&' + convertFormToString (this.formName);
 }

 // Get

 if (this.operation == 'GET') {
    getUrl = this.url + '?' + params;
    this.request.open ("GET", getUrl, true);
    this.request.onreadystatechange = this.callback;
    this.request.send (null);
    return (false);
 }

 // Post

 this.request.open ("POST", this.url, true);
 this.request.setRequestHeader ("Content-type", "application/x-www-form-urlencoded");
 this.request.setRequestHeader("Content-length", params.length);
 this.request.onreadystatechange = this.callback;
 this.request.send (params);
 return (false);
```

```
   }

   // Process Jt Messages

   this.processMessage = function (msg) {

        return (this.invokeAjaxRequest (msg));
   }
}

   // Process Jt Messages

   this.processMessage = function (msg) {


        return (this.invokeAjaxRequest (msg));

   }
}
```

The JtPortal reference application contains a Chat module (JtChat) that makes extensive use of AJAX . We'll use the Chat module to illustrate the AJAX integration.

The JtChat class encapsulates the JavaScript code. JtChat has several attributes. formName specifies the JSP form name. url specifies the Struts action to be invoked. These attributes are passed to the JavaScript component, JtAJAXAdapter. The sendChatMessage function for instance, sends the chat message via the JtAJAXAdapter. receiveChatMessages() receives chat messages. The receiveCallback function processes the AJAX reply. JtAJAXAdapter creates the AJAX request object used by the callback. The reply message (Jt.chat.Chat instance) is converted into XML.

```
// Chat class based on AJAX


function JtChat () {

  JtChat.timeout = 5000;
  this.keyboardEventDelta = 30000;
  this.formName='chatForm';    // HTML Form name
  this.url = '../AjaxChat.do'; // URL (Ajax request)


  // Default callback

  this.defaultAjaxCallback = function () {


  if (JtChat.sendRequest.readyState==4) {
      if (JtChat.sendRequest.status == 200) {

        jtReply = JtChat.sendRequest.responseText;

      }

   }

 }

  // Send a chat message

  this.sendChatMessage = function  () {
    var ajaxAdapter;
    var factory;


    factory = new JtFactory ();

    // Create JtAjaxAdater

    ajaxAdapter = new JtAjaxAdapter ();

    ajaxAdapter.url = this.url;                    // Struts action
    ajaxAdapter.formName=this.formName;            // JSP form
    ajaxAdapter.callback=this.defaultAjaxCallback;// AJAX callback

    // JtAjaxAdater creates the AJAX request object.
    // The callback uses this request object.

    JtChat.sendRequest = ajaxAdapter.request;

    factory.sendMessage (ajaxAdapter, new JtMessage ('JtSEND'));


    document.getElementById('message').value = '';

    JtChat.lastKeyboardEventDate=null;
    return (false);
  }


  // Receive callback

  this.receiveCallback = function() {
    var lastMessages;
    var statusMessage;
    var xmlDoc;

    if (JtChat.receiveRequest.readyState==4) {
      if (JtChat.receiveRequest.status == 200) {

        //window.alert (JtChat.receiveRequest.responseXML);

        if (JtChat.receiveRequest.responseXML == null) {
           mTimer = setTimeout('new JtChat().receiveChatMessages();',JtChat.timeout);
           return;
        }

        // A reply message (Jt.chat.Chat instance) is returned using
        // the XML format.

        xmlDoc=JtChat.receiveRequest.responseXML.documentElement;

        if (xmlDoc == null) {
           mTimer = setTimeout('new JtChat().receiveChatMessages();',JtChat.timeout);
           return;
        }

        // Retrieve the "lastMessages" attribute and update the chat window

        if (xmlDoc.getElementsByTagName("lastMessages")[0].childNodes.length > 0)
           lastMessages = xmlDoc.getElementsByTagName("lastMessages")[0].childNodes[0].nodeValue;
        else
           lastMessages = "";

        // Retrieve the "statusMessage" attribute and update the chat window

        if (xmlDoc.getElementsByTagName("statusMessage")[0].childNodes.length > 0)
          statusMessage =
             xmlDoc.getElementsByTagName("statusMessage")[0].childNodes[0].nodeValue;
        else
```

```
          statusMessage = "";


        document.getElementById ('statusMessage').innerHTML = statusMessage;
        if (lastMessages != "") {

           document.getElementById ('messageArea').value += lastMessages;
           textArea = document.getElementById ('messageArea');
           textArea.scrollTop = textArea.scrollHeight;

        }


        mTimer = setTimeout('new JtChat().receiveChatMessages();',JtChat.timeout);
      }

    }

  }

  // Receive chat messages

  this.receiveChatMessages = function () {

    factory = new JtFactory ();
    ajaxAdapter = new JtAjaxAdapter ();
    ajaxAdapter.url = this.url;                // Struts action
    ajaxAdapter.formName=this.formName;        // Struts form name
    ajaxAdapter.callback=this.receiveCallback; // callback
    JtChat.receiveRequest = ajaxAdapter.request;

    factory.sendMessage (ajaxAdapter, new JtMessage ('JtRECEIVE'));

  }

  // Invoke this function when the user starts typing a message.

  this.typingChatMessage = function () {

    var eventDate = new Date();
    var msg = new JtMessage ('KEYBOARD_EVENT');
    var timeDelta;


    if (JtChat.lastKeyboardEventDate != null) {

      timeDelta = eventDate.getTime() - JtChat.lastKeyboardEventDate.getTime();
      if (timeDelta < this.keyboardEventDelta) {

         return;
      }

    }

    JtChat.lastKeyboardEventDate = eventDate;

    ajaxAdapter = new JtAjaxAdapter ();
    factory = new JtFactory ();
    ajaxAdapter.url = this.url;
    ajaxAdapter.formName=null;
    ajaxAdapter.callback=this.defaultAjaxCallback;
    JtChat.sendRequest = ajaxAdapter.request;

    msg.msgData = document.getElementById('chatId').value;


    factory.sendMessage (ajaxAdapter, msg);

  }
```

```
   // Process Jt Messages
   //    JtSEND         - send a chat message
   //    JtRECEIVE      - receive chat messages
   //    KEYBOARD_EVENT - record a keyboard event so that other chat
   //                     users know that a message is being typed.


   this.processMessage = function (msg) {

      if (msg.msgId == 'JtSEND')
        return (this.sendChatMessage ());

      if (msg.msgId == 'JtRECEIVE')
        return (this.receiveChatMessages ());

      if (msg.msgId == 'KEYBOARD_EVENT')
        return (this.typingChatMessage ());

   }


}
```

The JSP (*JtPortal/Chat/Chat.jsp*) invokes the JavaScript componet JtChat:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:html="http://struts.apache.org/tags-html"
xmlns:bean="http://struts.apache.org/tags-bean"
xmlns:tiles="http://struts.apache.org/tags-tiles"
xmlns:c="http://java.sun.com/jstl/core" version="1.2">


<jsp:directive.page contentType="text/html; charset=UTF-8" />



<c:if test="${jtReply != null}">
  <c:set var="chatForm" value="${jtReply}" scope="request" />
</c:if>



<body onLoad="new JtChat().processMessage(new JtMessage ('JtRECEIVE'));">
  <div class="jtForumForm">
    <html:form action="Chat?msgId=JtSEND"
        onsubmit="var chat = new JtChat ();
        return chat.processMessage(new JtMessage ('JtSEND'));">
      <fieldset>
        <legend><bean:message key="Jt.chatForm.legend" /></legend>
        <br />
        <html:textarea property="messageArea" readonly="true" styleId="messageArea"
            cols="70" rows="10"></html:textarea>

        <br />
        <br />
        <label>
          <bean:message key="Jt.chatForm.message" />
          :
        </label>
        <html:text property="message" styleId="message" size="70"
            onkeypress="new JtChat().processMessage(new JtMessage ('KEYBOARD_EVENT'));"/>
        <html:hidden property="chatId" styleId="chatId" />
        <br />
        <br />

        <hr />


        <html:button styleClass="jtButton" value="Send" property="name"
           onclick="new JtChat().processMessage(new JtMessage ('JtSEND'));" />


      </fieldset>
    </html:form>
    <html:errors />
    <p id="statusMessage" > </p>

  </div>

</body>

</jsp:root>
```

The JavaScript Chat component invokes the Struts action AjaxChat.do. The following is the Struts Action mapping entry:

```
     <action path="/AjaxChat" name="chatForm"
       type="Jt.struts.JtStrutsAjaxAction"
       parameter="Jt.chat.ChatActionHandler"
       scope="request" validate="false" />
     <action path="/Chat" name="chatForm"
       type="Jt.struts.JtStrutsAction"
       parameter="Jt.chat.ChatActionHandler"
       scope="request" validate="false">
       <forward name="success" path="/Chat/Chat.jsp" />
       <forward name="failure" path="/Chat/Chat.jsp" />
     </action>
```

Notice that Jt.struts.JtStrutsAjaxAction needs to be employed when Struts are invoked via AJAX. Reply messages (Java objects) need to be converted to its XML representation before being returned to the JavaScript components. JtStrutsAjaxAction is a subclass of JtStrutsAction, the framework universal Jt action. Jt.chat.ChatActionHandler is the delegate Java class that will implement the chat logic.

The framework integration with AJAX also supports servlets. A single Jt servlet (Jt.servlet.JtServlet) provides an interface between AJAX and the Jt components. A sample application (*Jt/webapps/JtStrutsExamples*) illustrates the use of Ajax and JtServlet. To run this application, install JtStrutsExamples and open the browser to the appropriate URL http://localhost:8080/JtStrutsExamples (under Tomcat). The war file JtStrutsExamples.war can be downloaded from http://jt.dev.java.net (JtSampleApps.zip under documents & files). For additional information about JtServlet, please refer to the section dealing with the servlet integration.

13.1. JtServlet

To use the JtServlet you will need to add the following Servet definition to the web.xml file.

```
  <servlet>
    <servlet-name>JtServlet</servlet-name>
    <servlet-class>Jt.servlet.JtServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>JtServlet</servlet-name>
    <url-pattern>/JtServlet</url-pattern>
  </servlet-mapping>
```

Jt/webapps/JtStrutsExamples/ajax.jsp contains the url that invokes Ajax:

```
// Send the Ajax request to the JtAjaxServlet

function sendAjaxRequest() {

    ajaxAdapter = new JtAjaxAdapter ();

    factory = new JtFactory ();

    ajaxAdapter.url = "/JtStrutsExamples/JtServlet";

    ajaxAdapter.callback=callback;

    ajaxAdapter.classname = 'Jt.examples.HelloWorld'; // Delegate class

    ajaxAdapter.operation = 'GET';

    req = ajaxAdapter.request;

    factory.sendMessage (ajaxAdapter, new JtMessage ('JtHELLO'));

}

function callback() {

  if (req.readyState==4) {

    if (req.status == 200) {

      reply.value = req.responseText;

    }

  }

  clear ();

}
```

13.2. JtContext

The Java componets (JtStrutsAction and JtServlet) pass the Jt Context to the delegate class. The JtContext contains HttpServlet information (context, request, response, etc). This information may be required by the delegate class for the implementation of custom and advanced features. JtContext is part of the Jt package.

```
public class JtContext implements Serializable {

    public static final String JtCLASS_NAME = JtContext.class.getName();
    private static final long serialVersionUID = 1L;
    private Object servletContext;         // Servlet context
    private HttpServletRequest request;     // Servlet request
    private HttpServletResponse response;   // Servlet reponse

…


}
```

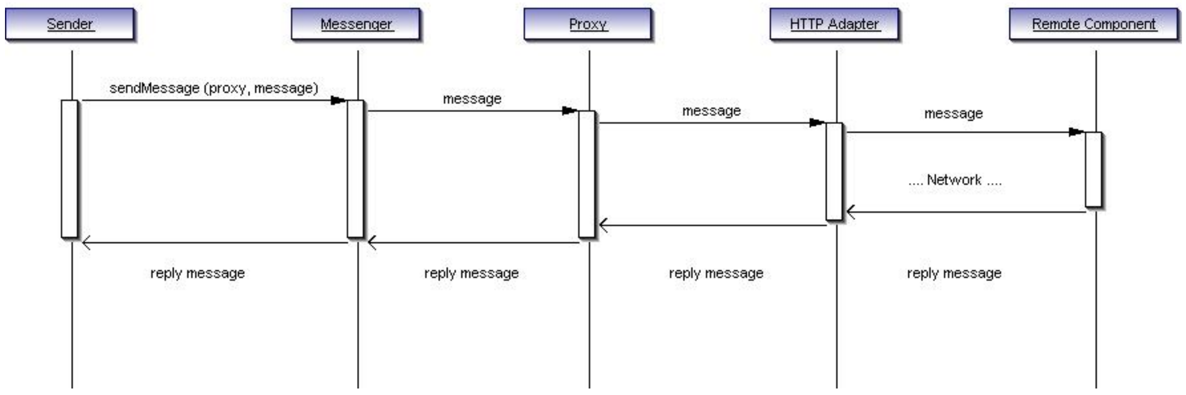
Several framework components illustrate the use of the Jt context. The list includes Jt.servlet.JtServlet and Jt.portal.Login.

14. Secure Web Services using MDP

14.1. Restful Web Services

REST stands for **Re**presentational **S**tate **T**ransfer. Jt framework components can be easily exposed as REST resources and services. For instance, the HelloWorld component can be accessed remotely using REST:

http://localhost:8080/JtPortal/JtRestService?classname=Jt.examples.HelloWorld&msgId=JtHELLO



Restful Web Service implementation using MDP

The REST integration is handled via the universal Jt servlet (Jt.servlet.JtServlet). As explained in previous sections, you will need to add the following Servet definition to the web.xml file:

```
  <servlet>
    <servlet-name>JtServlet</servlet-name>
    <servlet-class>Jt.servlet.JtServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>JtServlet</servlet-name>
    <url-pattern>/JtRestService</url-pattern>
  </servlet-mapping>
```

In general, every Jt component can be accessed using a REST request provided that access has been granted:

```
http://localhost:8080/JtPortal/JtRestService?classname=Jt.examples.DateService&msgId=DISPLAY_DATE
```

For security reasons, no access is granted unless you explicitly enable it by configuring the framework access manager (JtAccessManager):

Jt.security.JtAccessManager.classAccessList:Jt.examples.HelloWorld,Jt.service.*

This example grants remote access to Jt.examples.HelloWorld and the package Jt.service. For additional details please refer to the section dealing with the Jt Access Manager.

In summary, you will need to take a couple of simple steps to handle REST services and resources within the Jt pattern oriented framework:

a) Add the JtServlet to the web.xml file.
b) Configure the Jt Access Manager to grant access to the remote framework components.

This REST implementation is so simple thanks to the Jt messaging design pattern: ***public interface JtInterface { Object processMessage (Object message);}*** As explained earlier, this design pattern is the basis for the lego achitecture that provides transparent access to remote framework components. It reduces complexity by maximizing encapsulation and minimizing coupling. The JtRftp application relies on Restful web services. This reference application can be downloaded from the project page.

14.1.1. Jt Messaging information

As shown in the previous section, Jt messaging information needs to be passed to JtServlet. This component expects the following parameters:

a) jtMessage: message in XML format. If this parameter is present, the other parameters are ignored (msgId, msgContent, etc.). This parameter is optional. The conversion is usually done by the framework (please refer to the example).

or

a) msgId: Jt message Id (optional. If msgId is not present, JtACTIVATE is passed)

b) msgContent: Jt message content (optional)

c) msgData: Jt message data (optional)

d) msgAttachment: Jt message attachment (optional)

All these parameters are optional. They may be used to pass additional messaging data. JtServlet also expects an aditional parameter, classname: name of the remote component that will process the message. The remote class needs to be a Java bean and implement JtInterface. It should also declare and handle the attribute objException so that remote exceptions are propagated. If you prefer not to pass the className parameter, you can define subclasses of JtServlet and hard code the remote class. In this case, you will need one servlet per each class/service.

These parameters are consistent with the Jt messaging API. The reply object resulting from the execution of the delegate class is sent back to the client. Reply messages (Java objects of type other than java.lang.String) need to be converted to its XML representation before being returned to the client. This is automatically handled by JtServet. Java objects of type String are not converted. The Jt framework contains several helper classes that convert Java objects into XML and vice versa. These classes can be found under the package Jt.xml. For additional details, please refer to the section dealing with the XML integration. In our specific HelloWorld example, the reply is a String object ("Hello World …"). This string is returned to the client (no need for XML conversion).

The following invocation returns a Date instance:

```
http://localhost:8080/JtPortal/JtRestService?classname=Jt.examples.DateService&msgId=DISPLAY_DATE
```

In this case, the reply is converted to its XML representation:

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

<java version="1.5.0_06" class="java.beans.XMLDecoder">
 <object class="java.util.Date">
  <long>1257731352722</long>
 </object>
</java>
```

As usual the client application can run on any platform. The following sections of code illustrate a client written in Java using the Jt framework. It presents several advantanges:

a) Automatic conversion of the XML reply into a Java object.
b) Automatic handling and propagation of exceptions.

```
public static void main(String[] args) {

        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();
        JtMessage msg = new JtMessage (JtObject.JtACTIVATE);
        Object reply;
        JtRestService service;
        Date date;

        // Create an instance of JtRestService

        service = (JtRestService) factory.createObject (JtRestService.JtCLASS_NAME);

        // Assign the URL for the REST service/resource

        service.setUrl("http://localhost:8080/JtPortal/JtRestService");
        service.setClassname ("Jt.examples.HelloWorldMessage");

        // Send a message to the service. The message is converted into XML
        // and passed as jtMessage
        reply = messenger.sendMessage (service, "hi");

        // Check for exceptions. Remote exceptions are automatically
        // propagated by the framework.

        if (service.getObjException() == null)
                System.out.println ("Reply:" + reply);

        service.setUrl("http://localhost:8080/JtPortal/JtRestService");
        service.setClassname ("Jt.examples.DateService");
        msg = new JtMessage (DateService.GET_DATE);
```

```
                // The reply message is automatically converted to
                // the appropriate Java type by the framework

                date = (Date) messenger.sendMessage (service, msg);
                if (service.getObjException() == null)
                        System.out.println ("Reply date:" + date);
        }
```

14.1.2. Error and Exception Handling

As mentioned in the section dealing with the subject, the Jt framework provides a consistent/transparent way of handling errors and exceptions. The same mechanism applies to all the APIs integrated with the Jt Framework. This includes all the remote APIs. Exceptions generated by remote components are propagated to the local components. In the previous example, notice how remote exceptions are propagated to the local component (JtRestService).

Remote components are pretty much treated as local resources. The remote component doesn’t need to do anything special besides updating the objException and objErrors attributes appropriately. This is usually done by invoking handleException() or handleError(). Java beans that implement the Jt Interface will also need to declare and handle the objException attribute. For additional details, please refer to the section dealing with the Jt handling of exceptions and errors.

14.1.3. Secure Restful Web Services

As discussed in earlier sections, the framework provides strong security, encryption and authentication mechanisms. The following section of code enables encrypted/secure messaging between the local component and the remote component exposed as a REST service:

```
                JtMessenger messenger = new JtMessenger ();

              // Specify that secure/encrypted messaging should be used

                messenger.setEncrypted(true);

                service.setClassname ("Jt.examples.EchoService");

                msg = new JtMessage (JtObject.JtACTIVATE);
                sReply = (String) messenger.sendMessage (service, msg);
or

                service.setRemoteComponentId("jtEchoComponent");


                sReply = (String) messenger.sendMessage (service,
                       "Hello there ... Welcome to Jt messaging");
```

Messages are automatically encrypted before being sent to the remote component. The framework also automatically encrypts the reply message. This happens transparently thanks to the messaging design pattern: sender and recipient don't need to be aware that secure/encrypted messages are being transferred. Messages are also decrypted by the framework before they are processed by the recipient. For additional information, please refer to the sections dealing with framework security and access management. Jt.examples.RestServiceExample contains a complete example of web services invocation using secure/encrypted messaging. The following code illustrates message authentication. The authentication is done via JAAS and JtJAASAdapter. Username and password need to be passed as part of the message via a JtContext instance:

```
JtMessenger messenger = new JtMessenger ();
// Authenticated service
service.setUrl(url);
service.setClassname ("Jt.service.Logging");

// Encrypt the message (including username & password)
messenger.setEncrypted(true);

// Authentication information
context = new JtContext ();
context.setUserName("jt");
context.setPassword("messaging");

msg = new JtMessage (JtLogger.JtENABLE_LOGGING);
messenger.setContext(context);
reply = messenger.sendMessage (service, msg);
messenger.setContext(null);  // get rid of the authentication info
context.setPassword(null);
```

14.2. Axis Integration

Web services can be easily integrated within the Jt Framework. An MDP adapter provides a transparent interface between the Jt framework API and Web Services. The familiar Jt messaging API used to create and manipulate local framework objects still applies. Remote framework components are treated as local objects via a proxy. This section is based on the Apache Axis Web services implementation. The following simple steps are needed in order to invoke Axis web services:

A) Use a deployment descriptor to deploy the Jt Axis service (JtAxisService). The Jt Framework and Apache Axis need to be installed before performing this step.

B) Configure the Jt Access Manager to grant access to the remote framework components. Please refer to the section that deals with the Jt Access Manager.

C) Create an instance of the Axis Proxy (JtAxisProxy). This local proxy is used to manipulate the remote component. The JtAxisProxy component expects the following attributes:

   url: service url. This attribute is required.

   classname: class name of the remote component.

   componentId: Id of the remote component. Either classname or componentId needs to be specified. If componentId is specified, a component with the specified componentId must exist in the remote virtual machine (framework component registry). Otherwise an error will be produced: “Remote component not found.”

14.2.1. Deployment descriptor (deploy.wsdd)

The following is the web services deployment descriptor required by the Jt Framework. This deployment descriptor is Axis specific. Notice that only one deployment descriptor is required. This greatly simplifies the development and deployment of J2EE web services. The following descriptor deploys the Jt Axis service (JtAxisService class).

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

 <service name="JtAxisService" provider="java:RPC">
  <parameter name="className" value="Jt.axis.JtAxisService"/>
  <parameter name="allowedMethods" value="processMessage"/>
  <parameter name="scope" value="session"/>
 </service>

</deployment>
```

The following commands will deploy this descriptor:

```
cd src/Jt/axis

java org.apache.axis.client.AdminClient deploy.wsdd
```

14.2.2. Using the Jt Axis Proxy

Complete examples can be found under the source directory (Jt.axis.JtAxisProxy and Jt.examples.WebServicesExample). A local proxy is created. This proxy contains a reference to the remote component. The usual Jt messaging API can be used to manipulate the remote component via its proxy.

```
        /**
          * Demonstrates all the messages processed by JtAxisProxy.
          */

         public static void main(String[] args) {
           JtFactory factory = new JtFactory ();
           JtMessenger messenger = new JtMessenger ();
           String reply = null;
           Exception ex;
           JtMessage msg;
           JtAxisProxy proxy;
           Boolean Bool;

           // Create a local proxy that references a remote component

           proxy = (JtAxisProxy) factory.createObject (JtAxisProxy.JtCLASS_NAME);

           // Set the service url property (if it is not present in the resource file)
           if (proxy.getUrl() == null)
              proxy.setUrl("http://localhost:8080/axis/services/JtAxisService");
           proxy.setClassname(HelloWorldMessage.JtCLASS_NAME);


           factory.setValue (proxy, "greetingMessage", "Welcome to Jt messaging ...");
           System.out.println (factory.getValue (proxy, "greetingMessage"));

           // Send a message to the remote helloWorld component

           reply = (String) messenger.sendMessage (proxy, "hi");

           // Display the reply unless an exception is detected.
           // The remote exception gets propagated from the
           // remote object to the local proxy

           ex = (Exception) proxy.getObjException();

           if (ex == null)
             System.out.println (reply);

           // Remove the remote proxy

           factory.removeObject (proxy);

         }
}
```

```
/**
 * Demonstrates the use of the JtAxisProxy and the MDP distributed component model.
 * MDP provides transparent access to remote components via an Axis adapter.
 */

public class WebServicesExample  {
        public static final String GET_DATE = "GET_DATE";
        public static final String GET_DATE_TIME = "GET_DATE_TIME";
        public static void main(String[] args) {
        JtFactory factory = new JtFactory ();
        Date reply = null;
        Exception ex;
        Iterator iter;
        JtAxisProxy proxy;
        Boolean Bool;
        String sReply;
        String url = "http://localhost:8080/axis/services/JtAxisService";
        JtMessenger messenger = new JtMessenger ();
        Date date;
        List list;
        // Create a local instance of JtAxisProxy (references a remote component)

        proxy = (JtAxisProxy) factory.createObject (JtAxisProxy.JtCLASS_NAME);

        // Set the service url property (if it is not present in the resource file)
        if (proxy.getUrl() == null)
               proxy.setUrl("http://localhost:8080/axis/services/JtAxisService");
           proxy.setClassname("Jt.examples.DateService");

        // Send a message to the remote MDP component

        reply = (Date) messenger.sendMessage (proxy,
                new JtMDPMessage (WebServicesExample.GET_DATE));

        ex = (Exception) proxy.getObjException();

        // Display the reply unless an exception is detected

        if (ex == null)
            System.out.println ("Date:" + reply);

        // Request date and time

        list = (List) messenger.sendMessage (proxy,
                       new JtMDPMessage (WebServicesExample.GET_DATE_TIME));

        ex = (Exception) proxy.getObjException();

        if (ex == null) {

               iter = list.iterator();
               if (iter.hasNext()) {
                       date = (Date) iter.next();
                       System.out.println ("Date:" + date);

               }

               if (iter.hasNext())
                       System.out.println ("Time:" + iter.next());

        }
        // Remove the local proxy. The remote component is removed as well.
        factory.removeObject (proxy);
}
```

14.2.3. Invoking and Debugging the Web Services examples

Type one of the following commands in order to invoke the Web Services example:

```
java Jt.axis.JtAxisProxy
```

or

```
java –DLog=log.txt Jt.axis.JtAxisProxy
```

A hello message should be displayed. The second command will have logging/debugging information sent to log.txt. This only includes information associated with local objects. The service url is specified by the url attribute:

```
   proxy.setUrl ("http://localhost:8080/axis/services/JtAxisService");
```

This attribute can be added to the Jt resource file (Jt.properties):

```
    Jt.axis.JtAxisProxy.url: http://localhost:8080/axis/services/JtAxisService
```

Logging and debugging information generated by remote objects can be sent to the console or the remote file specified by the logFile attribute.

The second example (Jt.examples.WebServicesExample) can be invoked as follows:

```
   java Jt.examples.WebServicesExample
```

The JtRftp application relies on web services. This reference application can be downloaded from the project page.

14.2.4. Secure Web Services

As discussed in earlier sections, the framework provides strong security and encryption mechanisms. The following section of code enables encrypted/secure messaging between local component and remote components:

```
            proxy.setClassname ("Jt.examples.EchoService");
            messenger.setEncrypted(true);

            msg = new JtMessage (JtComponent.JtACTIVATE);
            sReply = (String) messenger.sendMessage (proxy, msg);
or

            proxy.setRemoteComponentId("jtEchoComponent");
            messenger.setEncrypted(true);

            sReply = (String) messenger.sendMessage (proxy,
                      "Hello there ... Welcome to Jt messaging");
```

Messages are automatically encrypted before being sent to the remote component. The framework also automatically encrypts the reply message. For additional information, please refer to the sections dealing with framework security and access management. Jt.examples.WebServicesExample contains a complete example of web services invocation using secure/encrypted messaging.

The following code illustrates message authentication. The authentication is done via JAAS and JtJAASAdapter. Username and password need to be passed as part of the message via a JtContext instance:

```
           proxy = (JtAxisProxy) factory.createObject
             (JtAxisProxy.JtCLASS_NAME);
            proxy.setUrl
             ("http://localhost:8080/axis/services/JtAxisService");
            proxy.setClassname("Jt.examples.LoggingService");
            // Encrypt the message (including username & password)
            // Authentication information
            context = new JtContext ();
            context.setUserName("jt");
            context.setPassword("messaging");
            messenger.setEncrypted(true);

            msg = new JtMDPMessage (JtLogger.JtENABLE_LOGGING);
            messenger.setContext(context);
            Bool = (Boolean) messenger.sendMessage (proxy, msg);
```

15. J2EE Design Pattern implementation using MDP

The Jt Framework implements several J2EE design patterns:

1) J2EE Service Locator

2) J2EE Session Façade

3) J2EE Business Delegate

4) J2EE Value Object

By using these design patterns, distributed framework applications are able to gain transparent access to remote components. This greatly simplifies the development of J2EE applications: no need to deal with many of the complexities of EJB development (deployment descriptors, home/remote interfaces, exception handling, etc.). The following steps are required in order to use the J2EE design patterns implemented by Jt:

A) The Jt distribution contains a JAR file (jtejb.jar) with the EJB classes and deployment descriptors. You will need to follow the installation instructions (Jt distribution) to deploy this JAR and the Jt Porperties file in your J2EE container/server. The client application also uses this jar file. A correct classpath needs to be set up for the application server and the EJB client application.

B) Create an instance of the J2EE Business Delegate class (JtBusinessDelegate). This class makes use of the J2EE Session Façade (JtSessionFacade) and the J2EE Service Locator (JtServiceLocator) classes.

C) Create a local instance of JtBusinessDelegate or JtEJBProxy. This local proxy is used to manipulate remote components. The Jt messaging API used for handling remote framework objects is the same. The access manager needs to be configured properly.

Documentation and sources for the J2EE components can be found at src/ejb and docs/ejbapi. . Examples to illustrate the use of the J2EE design patterns can be found under src/ejb/Jt/ejb/examples and jtapps:

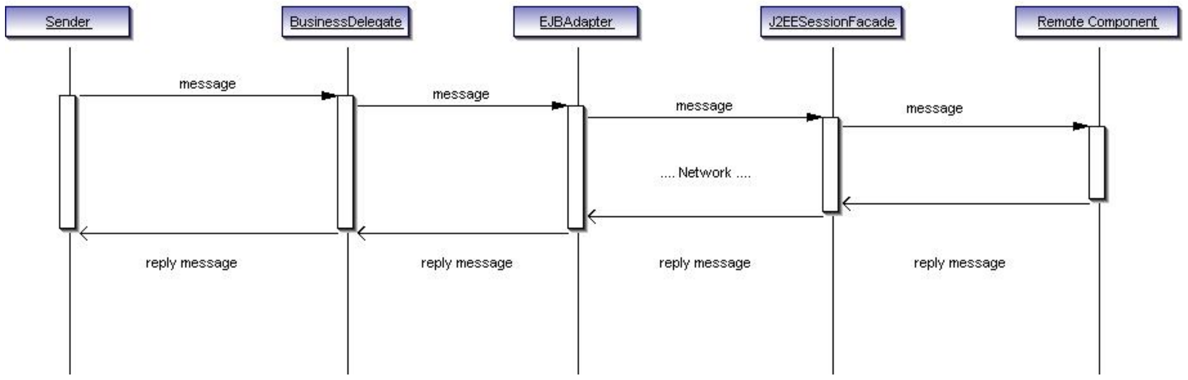
1) BusinessDelegateExample.java – example of J2EE Business Delegate. It uses J2EE Session façade and J2EE Service Locator.

2) ValueObject.java – example of J2EE Value Object.

3) EJBClient.java – complete example. It uses all the J2EE design patterns. It also demonstrates the use of BMP Entity Beans and Jt DAO objects. RemoteBusinessObject.java implements the business logic (remote framework

object). The complete ear file (ejbdemoapp.ear) can be downloaded from http://jt.dev.java.net (JtSampleApps.zip under documents & files).

When MDP is used, J2EE Session Façade is mainly responsible for forwarding the message to the appropriate remote component.

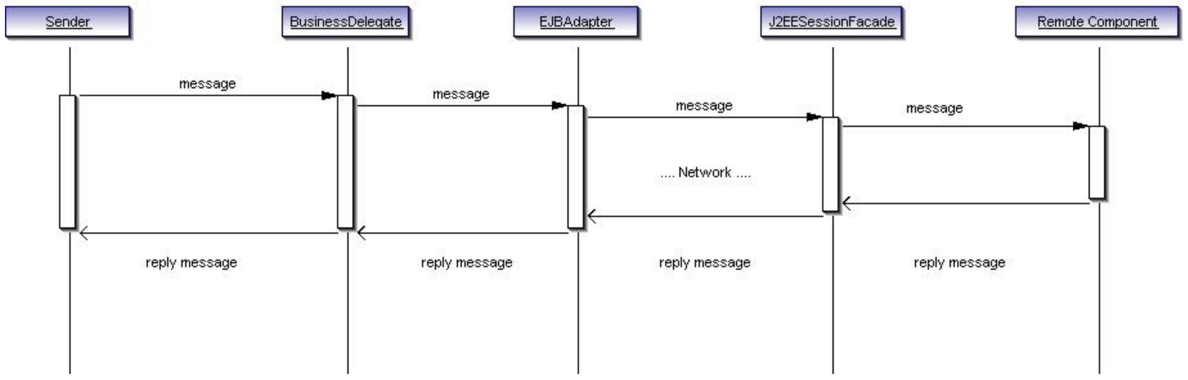


The following are the design patterns involved. For clarity sake the messenger component and the intrinsic processMessage() method have been removed from the UML diagram.

1) Business Delegate: the message is sent to the remote component via the business delegate.
2) EJBAdapter : adapter responsible for interfacing with the EJB API. It transfers the message to JtSessionFacade.
3) JtSessionFacade: forward the message to the appropriate remote component.

15.1. Using the Jt Business Delegate

When MDP is used, Business Delegate is mainly responsible for forwarding the message to the remote component via the EJBAdapter and J2EESessionFacade. The behavior is very similar to Proxy.



The following class illustrates the use of the Jt Business Delegate. The source code can be found at src/ejb/Jt/ejb/examples/BusinessDelegateExample.java.

```
/**

  * Demonstrates all the messages processed by JtBusinessDelegate.

  */

public static void main(String[] args) {

        JtFactory factory = new JtFactory ();
        JtFactory messenger = new JtMessenger ();
        JtBusinessDelegate businessDelegate;
        String tmp;
        String reply = null;
        Exception ex;



        // Create an instance of JtBusinessDelegate

        businessDelegate = (JtBusinessDelegate)
             factory.createObject (JtBusinessDelegate.JtCLASS_NAME);

        businessDelegate.setClassname(HelloWorldMessage.JtCLASS_NAME);


         factory.setValue (businessDelegate,
            "greetingMessage", "Hello there....");

         tmp = (String) factory.getValue (businessDelegate, "greetingMessage");

         System.out.println ("greetingMessage:" + tmp);


        // Send a message to the remote component

        reply = (String) messenger.sendMessage (businessDelegate, "hi");

        ex = (Exception) factory.getValue (businessDelegate, "objException");

        // Display the reply


        System.out.println ("reply:" + reply);


        // Remove the business delegate

        factory.removeObject (businessDelegate);

}
```

15.2. Invoking and Debugging the J2EE example

After installing the EJB components (EJB/jtejb.jar), type one of the following commands in order to invoke the J2EE example:

```
java Jt.ejb.examples.BusinessDelegateExample
```

or

```
java –DLog=log.txt Jt.ejb.examples. BusinessDelegateExample
```

A hello message should be displayed. The second command will have logging/debugging information sent to log.txt. This only includes information associated with local objects.

You will need to add jtejb.jar to the CLASSPATH. Under Windows, the command would look like the following:

```
java -classpath .\EJB\jtejb.jar;%CLASSPATH% Jt.ejb.examples.BusinessDelegateExample
```

The attribute values required by the EJB client application can be added to the framework properties file. Two files need to be updated (client and server). The following example uses Weblogic settings:

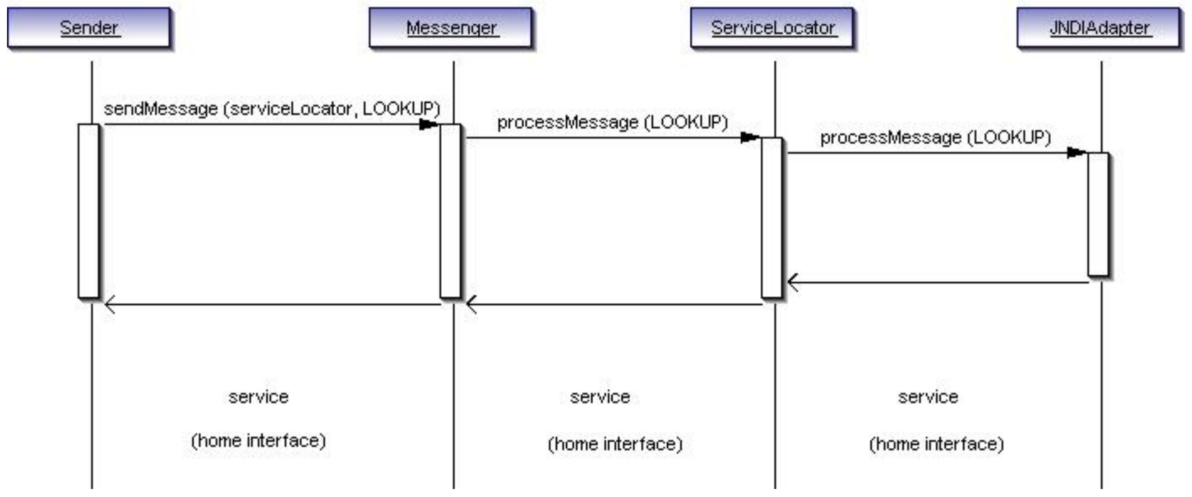
```
! Service Locator (Weblogic settings)

Jt.ejb.JtServiceLocator.url:t3://localhost:7001
Jt.ejb.JtServiceLocator.factory:weblogic.jndi.WLInitialContextFactory
!Jt.ejb.JtServiceLocator.user:weblogic
!Jt.ejb.JtServiceLocator.password:weblogic
```

The access manager (Jt.server.JtAccessManager) is the framework component responsible for granting remote access to framework components. Framework applications that use EJBs, JtServlet, REST, AJAX and Web Services rely on the access manager. For security reasons, no access is granted unless you explicitly enable it by configuring the framework access manager. This can be done by updating the framework properties file (server side):

```
Jt.security.JtAccessManager.classAccessList:Jt.examples.*,Jt.apps.RemoteBusinessObject
```

15.3. Using the Jt implementation of the J2EE Service Locator pattern

When messaging is used, Service Locator is mainly responsible for locating the service (home interface) by interfacing with the JNDI Adapter. JNDIAdapter is a messaging adapter that interfaces with the JNDI API.



The following sections of code illustrate the use of the Jt Service Locator (Jt.ejb.JtServiceLocator). There are basically four steps: a) Create an instance of the JtServiceLocator class. b) Set the appropriate attributes: EJBHome class and EJB's registered JNDI name in the case of EJBs c) Additional attributes may need to be changed depending on the J2EE application server. This includes factory, url, user, and password. The easiest way of making these changes is by updating the Jt properties file. Please refer to the installation instructions (EJB client section) d) Activate the object to locate the service by sending the JtACTIVATE message. An instance of the EJBHome class is returned (success) or null (failure). The Jt distribution contains complete source code and API documentation.

```
    /**
     * Demonstrates all the messages processed by JtServiceLocator.
     */

    public static void main (String[] args) {
        JtFactory factory = new JtFactory ();
        JtMessenger messenger = new JtMessenger ();

        JtServiceLocator locator;
        JtSessionFacadeHome jtservice;


        // Create an instance of JtServiceLocator

        locator = (JtServiceLocator) factory.createObject

                  (JtServiceLocator.JtCLASS_NAME);

        // Set the service locator attributes

        locator.setJndiName("JtSessionFacade");
        locator.setSclass(JtSessionFacadeHome.class);

        // Activate the Service Locator (locate the Service)

        jtservice = (JtSessionFacadeHome) messenger.sendMessage (locator,
           new JtMessage (JtObject.JtACTIVATE));


        if (jtservice != null)
            System.out.println  ("JtServiceLocator: GO");
        else
            System.out.println  ("JtServiceLocator: FAIL");



    }
```

15.4. Using the Jt implementation of the J2EE Value Object pattern

The Jt framework implements the J2EE Value Object design pattern via JtFactory which relies on component introspection to return a list of object attributes and their values. In general any remote proxy can be used to retrieve a Value Object via the JtGET_VALUES message:

```
        msg = new JtMDPMessage (JtFactory.JtGET_VALUES);
        factory.setValue(msg, JtMDPMessage.OBJECT, proxy);

        valueObject = (JtMDPValueObject) factory.processMessage(msg);
```

proxy is the reference to the remote component. A Value Object can be retrieved for all the APIs integrated with the framework including Web Services, J2EE EJB, Axis, HTTP, ESB, etc. Jt.JtMDPValueObject and Jt.examples.ValueObject illustrate the use of this design pattern. The following section is taken from Jt.examples.ValueObject:

```
package Jt.examples;
import java.util.*;
import Jt.JtFactory;
import Jt.JtMDPMessage;
import Jt.JtMDPValueObject;
import Jt.JtPrinter;
import Jt.http.JtHttpProxy;


/**
 * Demonstrates of the Jt implementation of the Value Object design pattern.
 */


public class ValueObject  {


    private static final long serialVersionUID = 1L;
    public static final String JtCLASS_NAME = ValueObject.class.getName();

    public ValueObject() {
    }


    public static void main(String[] args) {

        JtFactory factory = new JtFactory ();
        JtMDPMessage msg;
        JtMDPValueObject valueObject;
        JtPrinter printer = new JtPrinter ();
       JtHttpProxy proxy;
       //String url = "http://localhost:8082/JtServices/JtRestService";
       String url = "http://localhost:8080/JtPortal/JtRestService";
       Date date;
```

```
        // Create an instance of the remote Proxy

        proxy = (JtHttpProxy) factory.createObject (JtHttpProxy.JtCLASS_NAME);

        proxy.setUrl(url);
        proxy.setClassname ("Jt.examples.Test");


        date = new Date ();
        System.out.println("date:" + date);


        // Use the Proxy to update remote attributes

       factory.setValue(proxy, "date", new Date ());
       factory.setValue(proxy, "comments", "new comments");

        msg = new JtMDPMessage (JtFactory.JtGET_VALUES);
        factory.setValue(msg, JtMDPMessage.OBJECT, proxy);


        // Retrieve the value object via the proxy the remote API

        valueObject = (JtMDPValueObject) factory.processMessage(msg);

        if (valueObject == null) {
              System.out.println
               ("Unable to retrieve the value object:" + proxy.getObjException());
              System.exit(1);
        }

        // Retrieve the attribute values using the Value Object

        System.out.println
            ("date(ValueObject):" + factory.getValue(valueObject, "date"));
        System.out.println
            ("date(ValueObject):" + factory.getValue(valueObject, "comments"));

        printer.processMessage(valueObject);

    }

}
```

16. Component/XML conversion

XML is widely employed by the Jt framework. The Jt implementation of the Memento design pattern relies on XML to save and restore the state of an object. The framework also uses XML in order to interchange messages with remote components accessed via Web Services, AJAX and other APIs. The component/XML conversion is automatically handled by the Jt Framework. In other words, it is transparent to the application: from the application standpoint, there is no difference between messages sent to local or remote objects. Jt also offers capabilities to convert framework components into its XML representation and vice versa. The XML helper component (Jt.xml.JtXMLHelper) is responsible for handling the component/XML conversions. Several framework classes make use of Jt.JtXMLHelper. The list includes Jt.servlet.JtServlet, Jt.axis.JtAxisAdapter and Jt.struts.JtStrutsAction.

The following example illustrates the use of JtXMLHelper. The conversion is not limited to Jt framework objects. Although some restrictions apply, in general any Java object can be converted into XML and viceversa:

```
JtXMLHelper helper = new JtXMLHelper ();
String xmlString = (String) helper.processMessage (obj); // XML

Object obj = helper.processMessage (xmlString);
```

The standard messaging (with or without messenger involved) is equivalent:

```
JtMDPMessage msg = new JtMDPMessage (JtComponent.JtXML_DECODE);
factory.setValue(msg, JtMDPMessage.OBJECT, obj);

xmlString = (String) helper.processMessage(msg);

msg = new JtMDPMessage (JtComponent.JtXML_DECODE);
factory.setValue(msg, JtMDPMessage.OBJECT, xmlString);


obj = helper.processMessage(xmlString);
```

Arbitrary java objects can be converted using Jt.JtXMLHelper. The following section of code converts a complex object. Jt.JtXMLHelper contains many additional examples of XML/component conversion (JtComposite, List, subclasses of JtObject, all Java primitive types, etc.).

The framework offers additional XML capabilities via a DOM adapter (Jt.xml.JtDOMAdapter). The DAO adapter Jt.DAO.JtDAOAdapter illustrates the use of the JtDOM adapter.

17. Enterprise Service Bus (ESB) Messaging

The Jt Design Pattern Framework is also a MDP messaging engine that provides Enterprise Service Bus (ESB) capabilities. It features transparent access to components running inside remote applications. Framework components (local and remote) are able to interchange messages securely. The Jt framework also allows you to connect heterogeneous applications regardless of the technologies being used, including JMS, Web Services, EJB, REST, HTTP, EJBs, etc. Design patterns implemented by the framework (MDP, adapters, remote proxies, strategy, facades, etc.) make this possible. The Jt Enterprise Service Bus implementation consists of the following main components:

Enterprise Service Bus Adapter:Jt.esb.JtESBBridge

JMS Adapters: Jt.jms.JtJMSQueueAdapter and Jt.jms.JtJMSTopicAdapter

EJB Adapter: Jt.ejb.JtEJBAdapter

EJB Proxy: Jt.ejb.JtEJBProxy

Restful web services Adapter: Jt.rest.JtRestService

Secure web services Adapter (Axis): Jt.axis.JtAxisAdapter

Axis Proxy: Jt.axis.JtAxisProxy

Message encryption: Jt.security.JtMessageCipher

Message authentication: Jt.security.JtMessageAuthenticator

Access Manager: Jt.security.JtAccessManager

Data Access Objects: Jt.DAO.JtDAOStrategy

Java Mail Adapter: Jt.JtMail

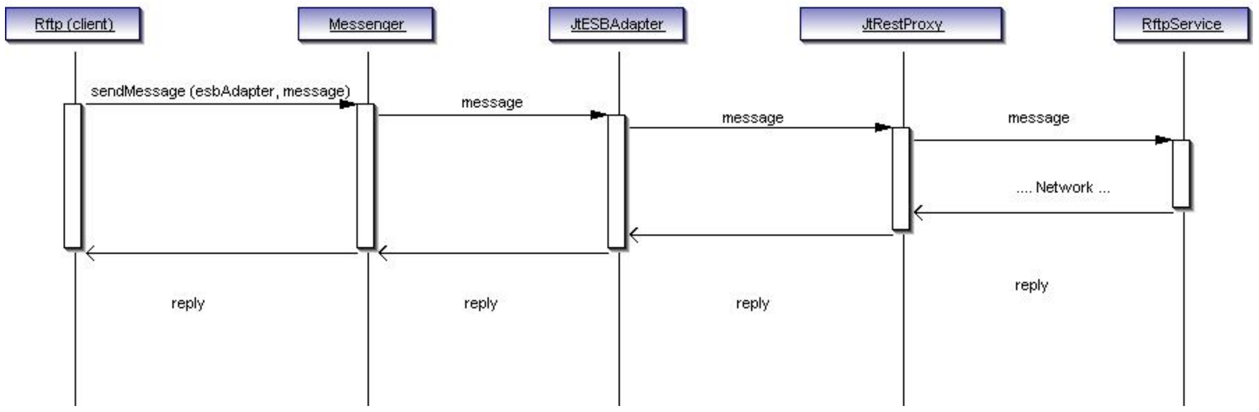
XML/Component conversions: Jt.xml.JtXMLHelper

Most of these components have been covered in earlier sections of this document. These components can be interchangeably plugged into complex framework applications using the “lego/messaging” architecture. They can be assembled in a variety of configurations to meet specific business requirements. In this case, these building blocks have been put together to provide Enterprise Service Bus (ESB) capabilities.

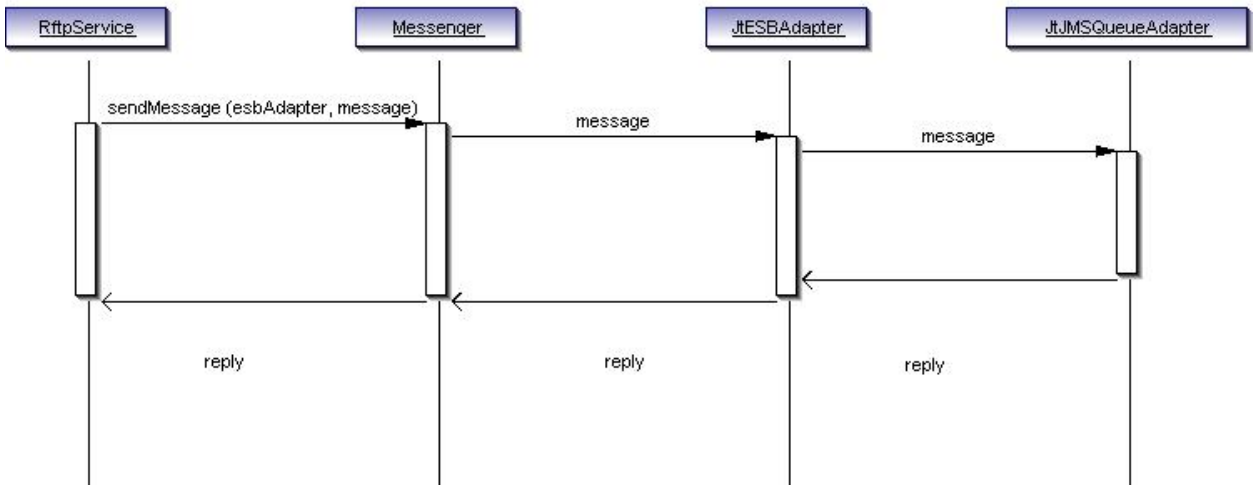
The framework ESB bridge connects applications to the Jt enterprise service bus. The ESB adapter can be configured to use any strategy for interchanging messages: JMS, secure Axis web services, EJBs, secure Restful web services, HTTP, etc. Custom/proprietary strategies and protocols may also be used. The ESB bridge and the other ESB components are also responsible for automatically converting the messages to the appropriate format/protocol. Synchronous, synchronous and secure messaging are supported

The framework ESB capabilities will be illustrated using one of the framework reference applications. The Reliable File Transfer Program (JtRftp) allows users to transfer files of any size reliably and securely by taking advantage of the framework Enterprise Service Bus capabilities and components. The following are the main applications and services involved:

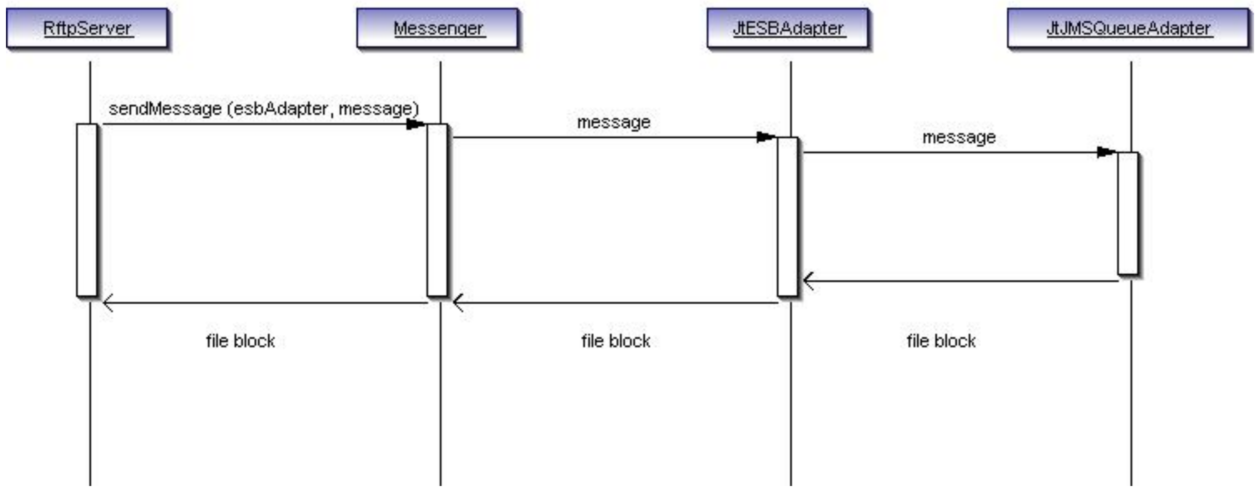
a) Rftp client application (Jt.ftp.Rftp). This client application transfers a file to a remote computer (one block of data at a time). It connects to the remote Rftp service component via the ESB adapter. The ESB adapter can be configured to use one of many available strategies to transfer the file to the server (Restful web services, Axis web services, EJBs, JMS, etc).



b) Rftp service component (Jt.service.RftpService). This remote component receives the file blocks being transferred and sends them to a JMS queue via the ESB adapter.



c) Rftp server application (Jt.rftp.RftpServer). This application retrieves the file blocks from the JMS queue via the ESB bridge. The use of JMS and transactions ensures the file transfer and makes the system reliable. Network failures don't have to prevent the transfer from completing. The client application retries to transfer the file block that failed. There is no need to retransmit the whole file. This is especially useful when using slow network connections and/or transferring very large files.



The complete JtRftp application can be downloaded from the project page. The following section of code illustrates how applications can connect to the framework ESB. As explained earlier, the ESB adapter can be configured to use one of many available strategies.

```
/**
  * Demonstrates the functionality provided by the ESB Adapter.
  */

 public static void main(String[] args) {

   JtFactory factory = new JtFactory ();
   JtESBBridge bridge;
   String url = "http://localhost:8080/JtPortal/JtRestService";
   String axisUrl = "http://localhost:8080/axis/services/JtAxisService";
   String sReply;
   JtMessenger messenger = new JtMessenger ();

   // Create an instance of JtESBBridge (Rest Strategy)

   bridge = (JtESBBridge) factory.createObject (JtESBBridge.JtCLASS_NAME);

   // Specify the implementor to be used by the ESB bridge.
   // Use the Restful web service.

   bridge.setImplementorClassname (JtRestService.JtCLASS_NAME);
   bridge.setUrl(url);

   // Specify the remote component
```

```
    bridge.setClassname (HelloWorldMessage.JtCLASS_NAME);

    sReply = (String) messenger.sendMessage (bridge, "hi");

    System.out.println("Reply:" + sReply);

    // Create an instance of JtESBBridge

    bridge = (JtESBBridge) factory.createObject (JtESBBridge.JtCLASS_NAME);

    // Specify the implementor to be used by the ESB bridge.
    // Use the Axis web service.

    bridge.setImplementor (new JtAxisProxy ());
    bridge.setUrl(axisUrl);

    bridge.setClassname (HelloWorldMessage.JtCLASS_NAME);   // Remote component

    sReply = (String) messenger.sendMessage (bridge, "hi");

    System.out.println("Reply(axis):" + sReply);

    // Create an instance of JtESBBridge (EJB strategy)

    bridge = (JtESBBridge) factory.createObject (JtESBBridge.JtCLASS_NAME);

    // Specify the implementor to be used by the ESB bridge.
    // Use the EJB strategy this time.

    bridge.setImplementor (new JtEJBProxy ());
    bridge.setClassname (HelloWorldMessage.JtCLASS_NAME); // Remote component

    sReply = (String) messenger.sendMessage (bridge, "hi");

    System.out.println("Reply:" + sReply);

    // Create an instance of JtESBBridge (EJB Strategy)

    bridge = (JtESBBridge) factory.createObject (JtESBBridge.JtCLASS_NAME);

    // Specify the implementor to be used by the ESB bridge.

    bridge.setImplementor (new JtEJBProxy ());
    bridge.setClassname (Echo.JtCLASS_NAME); // Remote Echo component
    System.out.println("Reply:" + "Welcome to Jt messaging ...");

    // Create an instance of JtESBBridge (JMS strategy)

    bridge = (JtESBBridge) factory.createObject(JtESBBridge.JtCLASS_NAME);

    JtJMSQueueAdapter jmsAdapter = (JtJMSQueueAdapter)
                  factory.createObject (JtJMSQueueAdapter.JtCLASS_NAME);

    // Specify the implementor (JMS strategy)

    bridge.setImplementor(jmsAdapter);

     messenger.sendMessage (bridge, "Hi there");
  }
```

The Rftp client application uses the following section of code in order to create an instance of the Enterprise Service Bus bridge.

```
esbBridge = (JtESBBridge)
        factory.createObject(JtESBBridge.JtCLASS_NAME, " esbRftpService");
```

Once the ESB bridge is created, the application can connect to the Jt Enterprise Service Bus and sent messages to remote applications/components in the usual way:

```
messenger.sendMessage(esbBridge, message);
```

Jt.properties should contain the appropriate properties:

implementorClassname: class name of the implementor to be used.

url: service url. This attribute is required when the strategy is REST or Axis.

classname: class name of the remote component.

componentId: Id of the remote component. Either classname or componentId needs to be specified. If componentId is specified, a component with the specified componentId should exist in the remote application (framework component registry). Otherwise an error will be produced: "Remote component not found."."

The following is the Jt.properties file associated with the JtRftp client application:

```
! ESB Adapter  (Restful service implementor)

#esbRftpService.implementorClassname:Jt.rest.JtRestService

#esbRftpService.url:http://localhost:7001/JtRftp/JtRestService

#esbRftpService.classname:Jt.service.RftpService
```

The strategy can be easily changed by using a different set of configuration values:

```
! ESB adapter (Axis implementor)

#esbRftpService.implementorClassname:Jt.axis.JtAxisProxy

#esbRftpService.url:http://localhost:8080/axis/services/JtAxisService

#esbRftpService.classname:Jt.service.RftpService
```

```
! ESB adapter (EJB implementor)

#esbRftpService.implementorClassname:Jt.ejb.JtEJBProxy

#esbRftpService.classname:Jt.service.RftpService
```

The JMS Adapter and the Jt Access Manager need to be configured via the appropriate Jt.properties file (Server side):

```
!Jt Access Manager (refer to the Access Manager section)

Jt.security.JtAccessManager.classAccessList:Jt.service.*


! JMS Adapter (point-to-point)

Jt.jms.JtJMSQueueAdapter.queue:testQueue

Jt.jms.JtJMSQueueAdapter.connectionFactory:TestJMSConnectionFactory

Jt.jms.JtJMSQueueAdapter.timeout:1
```

18. Android

Because of MDP, the Java Design Pattern framework is efficient and lightweight. It is able to run on smartphones under Android. Several Android projects are part of the Jt distribution. Using the Jt framework as part of your Android project is a very straightforward task. Under Eclipse, you simply need to add the Jt project to the build path. You may also add the Jt jar (Jt.jar) to the build path. Please keep in mind that some of the framework functionality won't be supported, because Android is not based on the standard Java SDK. The following functionality is supported:

a) Messagin Design Pattern (MDP) and Jt core framework functionality.

b) GoF Design Pattern implementation based on MDP.

c) Jt secure web services (including Restful services). The Axis components are not supported under Android.

d) Logger component.

e) Jt DAO and JDBC Adapter. The Hibernate components are not supported under Android.

f) BPEL engine and components.

g) Component/XML conversion functionality.

h) J2EE design patterns and components are not supported under Android (JMS, EJBs, etc).

i) Limited ESB functionality because J2EE and Axis components are not supported.

Two Android/Eclipse projects demonstrate the Android integration:

1) HelloAndroid: demonstrates Android Web Services.

2) DAOAndroid: demonstrates Android Data Access Objects (DAOs) and the JDBC Adapter (JtJDBCAdapter). Please consult the section dealing with JDBC and DAOs.

3) Jtandroid: MDP components for Android. These reusable MDP components implement GPS capabilities, voice recognition, GUI, Text-To-Speech (TTS), Media Player, SMS, etc.

Notice that Android can access all the framework functionality via remote proxies and web services. The following example illustrates an Android application.

```java
package com.example.helloandroid;


import Jt.JtFactory;
import Jt.JtMessenger;
import Jt.examples.HelloWorldMessage;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;



/*
 * Android demo program based on the Jt Design pattern framework.
 */

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
      String reply;
      JtFactory factory = new JtFactory ();
      JtMessenger messenger = new JtMessenger ();
      HelloWorldMessage helloWorld;

        super.onCreate(savedInstanceState);

        TextView tv = new TextView(this);


        // Create an instance of HelloWorldMessage

        helloWorld = (HelloWorldMessage)
          factory.createObject(HelloWorldMessage.JtCLASS_NAME);

        // Send the Message

        reply = (String) messenger.sendMessage(helloWorld, "hi");

        tv.setText("Hal: " + reply);
        setContentView(tv);

    }
}
```

The following example illustrates the invocation of a MDP Restful service. Notice that under Android the attributes *android* and *propertiesPath* are required in order to load Jt.properties. The Jt properties file should contain the following entry:

```
! Android Properties

Jt.xml.JtXMLHelper.parserName:org.xmlpull.v1.sax2.Driver
Jt.xml.JtSAXAdapter.parserName:org.xmlpull.v1.sax2.Driver
```

This entry is required for XML processing and Restful web services. Besides these minor configuration paramenters, under the Jt framework, everything else should work the same regardless of the target platform (Andriod or any other Java platform). This is another advantage of using Jt.

```
package com.example.helloandroid;


import Jt.JtFactory;
import Jt.JtJDBCAdapter;
import Jt.JtMessage;
import Jt.JtMessenger;
import Jt.JtObject;
import Jt.http.JtHttpProxy;
import Jt.xml.JtXMLHelper;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;


/*
 * Android demo program based on the Jt Design pattern framework.
 */

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
      String reply;
      JtFactory factory = new JtFactory ();

        factory.setAndroid(true); // Android is being used.

        // Jt properties file. Jt.properties needs to be copied
        // to the appropriate directory


        factory.setPropertiesPath("/com/example/helloandroid/Jt.properties");

        super.onCreate(savedInstanceState);


        TextView tv = new TextView(this);


        // Invoke a Restful MDP service

        reply = invokeService ();

        tv.setText("Hal: " + reply);
        setContentView(tv);
    }





      /*
       * Invoke a Restful MDP service
       */

    private String invokeService () {
            JtFactory factory = new JtFactory ();
            String sReply;

            JtHttpProxy proxy;

            String url =

            "http://freedom.lunarpages.com/JtServices/JtRestService";
            JtMessenger messenger = new JtMessenger ();


            // Create an instance of the remote Proxy

            proxy = (JtHttpProxy)
               factory.createObject (JtHttpProxy.JtCLASS_NAME);

            proxy.setUrl(url);
            proxy.setClassname ("Jt.examples.Echo");


            sReply = (String)
              messenger.sendMessage (proxy, "Welcome to MDP messaging ...");

            if (sReply == null) {
                  if (proxy.getObjException() != null)
                        return (proxy.getObjException().toString());
            }
            return (sReply);

    }
```

Under Android, the Jt components (DAO, JDBC, secure messaging, etc.) need to be appropriately configured via Jt.properties. Please consult the installation notes for additional details. The following are some examples:

```
! JDBC Adapter (SQLite/Android settings)


Jt.JtJDBCAdapter.driver:SQLite.JDBCDriver
Jt.JtJDBCAdapter.url:jdbc:sqlite:/sdcard/jttest.db
```

```
! DAO Adapter

Jt.DAO.JtDAOAdapter.configFile:/com/example/dao/hibernate.cfg.xml
Jt.DAO.JtDAOAdapter.driver:SQLite.JDBCDriver
Jt.DAO.JtDAOAdapter.url:jdbc:sqlite:/sdcard/jttest.db

! Jt KeyStore (client application)

Jt.security.JtKeyStore.password:password
Jt.security.JtKeyStore.alias:Jt
Jt.security.JtKeyStore.resourcePath:/com/example/helloandroid/JtClient.ks


! Jt Asymmetric Cipher

Jt.security.JtAsymmetricCipher.certificateName:Jt
Jt.security.JtAsymmetricCipher.transformation:RSA/ECB/PKCS1Padding


! Jt Symmetric Cipher

Jt.security.JtSymmetricCipher.algorithm:AES
Jt.security.JtSymmetricCipher.transformation:AES
Jt.security.JtSymmetricCipher.keySize:128
```