Bean Markup Language (Version 2.3)

User's Guide

Sanjiva Weerawarana and Matthew J. Duftler
IBM TJ Watson Research Center
Hawthorne, NY 10532.

September 22, 1999

http://www.alphaWorks.ibm.com/formula/bml
mailto:bmlers@watson.ibm.com

ABSTRACT

Bean Markup Language (BML) is an instance of an XML-based component configuration or wiring language customized for the JavaBean component model. The language is designed to be directly executable; i.e., processing a BML script will result in a running application configured as described in the script. The BML language has elements that can be used to describe the creation of new beans, accessing of existing beans, configuration of beans by setting/getting their properties and/or fields, binding of events from beans to other beans, as well as calling arbitrary methods in beans.

We provide two implementations of BML - a player which interprets a BML script to produce a running bean and a compiler which translates a BML script to Java. The player is implemented using Java reflection and is very small (approximately a 70K jar file). The compiler produces reflection-free Java code similar to the code one would write by hand to implement the same component structure described in the BML script it is compiling. The advantage of using BML and the compiler to generate the Java code vs. writing the code directly is that the machine generable BML language captures in a first-class form the inter-component structure of the application which can then be compiled away to avoid any run-time performance loss.

This document is the user's guide for the BML language, player, and compiler.

Table of Contents

1. Introduction

BML is an XML language for describing the structure of a set of interconnected beans. It is not an XMLized Java syntax - it can only be used to describe how a set of beans are to be created, configured and interconnected. It is in many ways a new kind of language - it is neither a Turing-complete programming language nor a full scripting language; it is a language whose only functions are to describe how components relate to one another and how each of those components is configured. It is however, not a modeling language - it is directly executable.

1.1 BML Design Goals

The following goals were conscientiously observed during the design and development of BML:

- BML's language elements will be a minimal basis of operators necessary to configure any set of beans (i.e., any Java object should be operable on by BML)
- BML will not make any assumptions about the types of beans it can configure
- BML will not make any assumptions about the types of events that beans may throw and it will not provide any preferential or built-in support for any specific set of event types
- BML will not make any assumptions about the data types of property and field values
- BML will not make any assumptions about the types of containers it can work with in its hierarchy creation capability
- BML implementations will be easily embeddable in other software
- BML created/configured beans will interact well and easily with beans created/configured by other means
- BML architecture and implementations will be extensible to allow the addition of new wiring capabilities

1.2 Required User Background

BML users are expected to have a working knowledge of XML, and a good understanding of the JavaBeans component model.

2. The BML Language

The BML language has been carefully designed to be easily machine generated as well as hand-crafted. The language provides a first-class mechanism for capturing the structure of a component application, versus the Java language which loses some of this information in the syntax of the language. In addition, since BML is XML-based, it opens up application development to all the tools now being developed to work with XML. For example, you can start to think about using XSL to "style" applications, starting from an XML description of the program's functions (or an individual user's needs) and an XSL description of a user-interface framework, and generating BML to produce a running application. (In fact, this idea is what got us started on the BML project in the first place!)

BML allows one to specify the desired configuration of a set of beans. This configuration is expressed in terms of standard bean operations, including creation, event bindings, method calls and containment. These elements are designed to be a concise, orthogonal set of operations with which one is able to specify any desired bean configuration. The following table briefly defines the BML language elements:

Element	Description
<bean>	Create a new bean or look one up
<args>	Specify constructor arguments
<string>	Create a new string bean or look one up
<property>	Set or get a bean property
<field>	Set or get a bean field
<event-binding>	Bind an event from one bean to another
<call-method>	Call a bean method
<cast>	Type convert a bean to be of another type
<add>	Create a hierarchy of beans by adding one to another
<script>	Defines a (BML or other) script to be used somewhere

BML processors also support a single processing instruction which can be used to load components to the processor's runtime.

In this section we describe each of the elements in BML and their interactions with one another.

2.1 Bean Creation and Access

The <*bean*> element is used to create new beans or to look up beans by name. Once the bean has been created or looked up, the children of the bean element are processed with this bean as the default target of the operations performed by the children. That is, if a <*call-method*> (see below) element were present as a child, for example, unless otherwise stated in the <*call-method*> element, the call would occur on the bean identified by the containing <*bean*> element. If there are no children elements, then processing the <*bean*> element simply results in that bean.

Note: Even if a child element changes its target bean by using the target attribute, the *default* target bean remains the bean identified by the nearest enclosing <*bean*> element. For example,

```
<bean class="Foo" [id="foo-bean"]>
  <property target="bar-bean" name="p1">
    <property name="potato"/>
  </property>
</bean>
```

would result in the p1 property of bar-bean being set to the value of the potato property of foo-bean.

2.1.1 Creating a Bean

Creating a bean may result in the resurrection of a bean from a serialized file (a ".ser" file), the instantiation of a new instance of a class, or the processing of an inner BML file. Regardless of the method of creation, the bean may optionally be registered into BML's object registry which provides a name to object reference mapping (see BML Processing and Run-time Environments sections). The different cases of using the <*bean*> element are listed below.

Case 1: Creating a bean by instantiating a class using its no-args constructor or by resurrection, and optionally registering it:
Syntax:

```
<bean class="class-name-or-.ser-file-name" [id="name-to-register-as"]>
  ... other BML elements to configure bean ...
</bean>
```

Semantics: A new instance is created by either locating a serialized file (using the class loader in the current BML environment (see Section 4.1)) with the given name or by creating a new instance from the class using its no-args constructor. If the "id" argument is present, the bean is registered into the object registry with that name as its key.

Examples:

```
<bean class="java.awt.Panel" id="topPanel">
  <add>
    <bean class="java.awt.Label">
      <property name="text" value="Demo Label"/>
    </bean>
  </add>

  <add>
    <bean class="mySavedButton"/>
  </add>
</bean>
```

Case 2: Creating a bean by instantiating a class with a specific constructor and optionally registering it:
Syntax:

```
<bean class="class-name" [id="name-to-register-as"]>
  <args>
    constructor-arg-1
    constructor-arg-2
    ...
```

```
    constructor-arg-n
  </args>
  ... other BML elements to configure bean ...
</bean>
```

Semantics: Each argument is evaluated using as the default target the bean containing the bean being created. After loading the class, a constructor whose signature matches the types of the arguments (using the algorithm defined by the Java language specification) is located and used to instantiate the bean. If the "id" argument is present, the bean is registered into the object registry with that name as its key.

Examples:

```
<bean class="java.awt.Dimension">
  <args>
    <cast class="int"><string>200</string></cast>
    <cast class="int"><string>100</string></cast>
  </args>
</bean>
```

Case 3: Creating a bean by processing an inner BML file and optionally registering its return value:
Syntax:

```
<bean class="URL-or-filename-of-BML-file" [id="name-to-register-as"]>
  [<args>
    script-arg-1
    script-arg-2
    ...
    script-arg-n
  </args>]
  ... other BML elements to configure bean ...
</bean>
```

Semantics: If an args element is present as the first element child of the *<bean>* element, then each argument is evaluated using as the default target the bean containing the bean being created. Then, the values of the arguments are registered using the special names "script:arg0", "script:arg1", etc. in a special registry that is available to the inner BML file. The inner file is then processed, and the return value is optionally registered. The inner file is processed in the following manner:

- The inner file will use as its default target the bean containing the bean being created.
- All lookups will cascade upwards.
- All registrations will be done locally. (The only value from the inner file that will be available to the outer file is the return value; that is, the outermost containing element of the inner file.)
- Inner BML files are processed basically the same way *<script>* elements are.

Examples:

```
<bean class="http://bml.watson.ibm.com/repos/fooFrame.bml">
  <args>
    <string value="TestFrame"/>
  </args>

  <add>
    <bean class="barTextField.bml"/>
  </add>
</bean>
```

Explanation: The content of http`://bml.watson.ibm.com/repos/fooFrame.bml` is retrieved and processed. A string with the value "TestFrame", referred to by the name "script:arg0", will be available while processing the `fooFrame.bml` script. Then, the content of

`http://bml.watson.ibm.com/repos/barTextField.bml` will be retrieved and processed; its return value will be added to the outermost <*bean*>.

Note: When the value of the class attribute has the extension ".bml", it is first treated as a URL. If the value is not a valid URL as specified, the contextURL (see section 3.1.1) is used as a base. If this too fails, a final attempt is made at resolving the location by treating the value as a file name. Some sample values for the class attribute are: `http://bml.watson.ibm.com/demos/atool.bml`, `d:\temp\app.bml`, `myCustomMenuBar.bml`, `../myHelpMenu.bml`.

2.1.2 Looking Up a Bean

The <*bean*> element also allows one to look up beans previously registered in the object registry. A special feature in the bean element allows one to look up the java.lang.Class object representing some Java class. The different cases of using the <*bean*> element are listed below.

Case 1: Looking up a previously registered bean:
Syntax:

```
<bean [class="class-of-registered-bean"] source="previously-registered-name">
  ... other BML elements to configure bean ...
</bean>
```

Semantics: The name is looked up in the object registry and if found that bean is used. The “class” attribute is optional, but required if this BML script is being compiled and if the bean itself is unavailable at compile-time. In this case, without this information the BML compiler would have no information about the bean in order to compile the script. If the class attribute is specified on a lookup, then it must match the class of the bean; otherwise an exception is thrown.

Examples:

```
<bean source="topPanel">
  ... other BML elements to configure bean ...
</bean>

<bean source="myBean" class="class-of-myBean">
  ... other BML elements to configure bean ...
</bean>
```

Case 2: Looking up a class bean:
Syntax:

```
<bean source="class:name-of-class-to-find">
  ... other BML elements to configure bean ...
</bean>
```

Semantics: The class of name “name-of-class-to-find” is loaded and the instance of java.lang.Class object representing it is returned. Special recognition has been given to the names “boolean”, “byte”, “char”, “short”, “int”, “long”, “float” and “double” to enable access to the class objects representing the primitive types. For example, setting the source attribute to “class:int” would retrieve the object int.class. This can be used to make static method calls.

Examples:

```
<bean source="class:java.lang.System">
  <call-method name="currentTimeMillis"/>
</bean>

<bean class="java.awt.Button">
  <event-binding name="action">
    <script>
```

```
      <call-method target="myActionHandler" name="doIt">
        <bean source="bml:arg1"/>
      </call-method>
    </script>
  </event-binding>
</bean>
```

2.2 Creating String Beans

String beans (i.e., instances of java.lang.String), unfortunately, have to be treated specially because of both Java and XML. Java strings are immutable objects - so you have to create a string with its value and live with it forever. That means that the only way to create a (non-empty) string is to use a <*bean*> element to create it by invoking a specific constructor which requires a string argument (see case 2 of Section 2.1.1). Syntactically that would be approximately:

```
<bean class="java.lang.String">
  <args>... something to give value..</args>
</bean>
```

The question now becomes how does one indicate the string value in BML? One option is to make the <*args*> element be of mixed content - i.e., it has other elements (such as <*bean*> etc. which are needed to call arbitrary constructors) as well as literal text. Syntactically that would be approximately:

```
<bean class="java.lang.String">
  <args>Hello There</args>
</bean>
```

This would work reasonably well, except that XML is whitespace sensitive. Hence, if one were to (seemingly) reformat the above script as shown below, the behavior would be different because the value of the string would be "\n Hello There\n " instead of just "Hello There":

```
<bean class="java.lang.String">
  <args>
    Hello There
  </args>
</bean>
```

The situation is worse - suppose that one wished to invoke a constructor that required two strings. This approach would simply not allow that case to be handled.

The solution that BML takes is to introduce a new element, the <*string*> element. This element is used to create string beans. There are two cases of usage:

Case 1: Creating a string with a non-empty value:
Syntax:

```
<string [value="value-of-string"]>[value-of-string]</string>
```

Semantics: An instance of class java.lang.String is returned with the value "value-of-string". If the value attribute is given, the the value of that attribute is used as the value of the string. If not, the entire contents between the open and close tags are used as the value (including any whitespace). The content of <*string*> is mixed XML content; i.e., it consists of any number of inter-mixed CDATA sections and text.

Examples:

```
<bean class="java.awt.Dimension">
  <args>
    <cast class="int"><string>200</string></cast>
    <cast class="int"><string value="100"/></cast>
  </args>
</bean>
```

Case 2: Creating an empty string:
Syntax:

```
<string/>
```

Semantics: This is entirely equivalent to *<bean class="java.lang.String"/>*.

Examples:

```
<bean source="class:java.lang.System">
  <call-method name="getProperty">
    <string>user.name</string>
    <string/>
  </call-method>
</bean>
```

2.3 Bean Property Configuration

The *<property>* element allows one to get the value of a property or to set the value of a property to some value. The value may be immediate or the return value of some other appropriate BML element. The property setting or getting is performed on the default target bean (i.e., the bean identified by the *<bean>* element within which this element is contained), unless otherwise specified in this element. Properties which are of primitive types (int, float etc.) are also supported by the *<property>* element. The different cases of using the *<property>* element are listed below.

Case 1: Setting a property value with an immediate value:
Syntax:

```
<property [target="alternate-bean"] name="property-name" [index="num"]
          [id="name"] value="property-value"/>
```

Semantics: This is a syntactic short-cut for:

```
<property [target="alternate-bean"] name="property-name" [index="num"]
          [id="name"]>
  <string>property-value</string>
</property>
```

See the case below for its semantics.

Examples:

```
<bean class="java.awt.Label">
  <property name="text" value="Hello There!"/>
</bean>
```

Case 2: Setting a property value with an indirect value:
Syntax:

```
<property [target="alternate-bean"] name="property-name" [index="num"]
          [id="name"]>
  ... a bean, string, field, property, call-method, cast or script element ...
</property>
```

Semantics: Sets the value of the property to the given value. The value is obtained by evaluating the single child element and obtaining the return value (bean). If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then setting the value of the property. If the index attribute is present, then property must be indexed and the num'th indexed property is set. If the type of the value and the type of the property do not match (or are not assignable), a type conversion will be necessary. For example, setting the background color of an AWT component to blue would require a convertor to convert the string "0x0000ff" to the appropriate java.awt.Color object representing that color. BML uses a registry of available type convertors (*TypeConvertorRegistry*) which provides a reference to a *TypeConvertor* object that can handle the necessary conversion. The convertor is dynamically invoked with the value to be converted and the result used as the value of the property. Type convertors and the type convertor registry are discussed in Section 4.3. If the id attribute is present, then the value is registered in the object registry with that name.

Examples:

```
<bean class="java.awt.Frame">
  <property name="layout">
    <bean class="java.awt.BorderLayout"/>
  </property>
</bean>
```

Case 3: Getting a property value:
Syntax:

```
<property [target="alternate-bean"] name="property-name" [index="num"]
          [id="name"]/>
```

Semantics: Returns the value of the named property as a bean. If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then getting the value of the property. If the index attribute is present, then property must be indexed and the num'th indexed property value is retrieved. If the id attribute is present, then the value is registered in the object registry with that name.

Examples:

```
<bean class="java.awt.Label">
  <property name="text">
    <property target="myDataField" name="labelString"/>
  </property>
</bean>
```

2.4 Bean Field Configuration

The <*field*> element allows one to get the value of a field or to set the value of a field to some value. The value may be immediate or the return value of some other appropriate BML element. The field setting or getting is performed on the default target bean (i.e., the bean identified by the <*bean*> element in which this element is contained), unless otherwise specified in this element. Static fields may be manipulated by setting the target bean to the class object which owns the field (see Section 2.1.2). The <*field*> element semantics are almost exactly the same as those of the <*property*> element and are included here for the sake of completeness. The primary difference between <*field*> and <*property*> is that fields may be static. Fields which are of primitive types (int, float etc.) are also supported by the <*field*> element. The different cases of using the <*field*> element are listed below.

Case 1: Setting a field value with an immediate value:
Syntax:

```
<field [target="alternate-bean"] name="field-name" [id="name"]
       value="field-value"/>
```

Semantics: This is a syntactic short-cut for:

```
<field [target="alternate-bean"] name="field-name" [id="name"]>
  <string>field-value</string>
</field>
```

See the case below for its semantics.

Examples:

```
<bean class="myClass">
  <field name="size" value="100"/>
</bean>
```

Case 2: Setting a field value with an indirect value:
Syntax:

```
<field [target="alternate-bean"] name="property-name" [id="name"]>
```

```
    ... a bean, string, field, property, call-method, cast or script element ...
  </field>
```

Semantics: Sets the value of the field to the given value. The value is obtained by evaluating the single child element and obtaining the return value (bean). If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then setting the value of the field. If the type of the value and the type of the field do not match (or are not assignable), a type conversion will be necessary. BML uses a registry of available type convertors (*TypeConvertorRegistry*) which provides a reference to a *TypeConvertor* object that can handle the necessary conversion. The convertor is dynamically invoked with the value to be converted and the result used as the value of the field. Type convertors and the type convertor registry are discussed in Section 4.3. If the id attribute is present, then the value is registered in the object registry with that name.

Examples:

```
  <bean class="myClass">
    <field name="size">
      <string>100</string>
    </field>
  </bean>
```

Case 3: Getting a field value:
Syntax:

```
  <field [target="alternate-bean"] name="property-name" [id="name"]/>
```

Semantics: Returns the value of the named field as a bean. If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then getting the value of the field.

Examples:

```
  <bean class="java.awt.Label">
    <property name="alignment"> <!-- center the label -->
      <field target="class:java.awt.Label" name="CENTER"/>
    </property>
  </bean>
```

2.5 Bean Event Binding

The <*event-binding*> element supports binding of events from a bean to something else. The JavaBeans event model states that if a bean ("source") can generate an event of type XEvent, then any listener ("target") must implement the XListener interface. Beans that implement the listener interface for an event can be registered as a listener of that event at the source bean. The listener interface defines a set of methods via which the event may be delivered by the event source to the event listener. The <*event-binding*> element supports both this form of event binding as well as binding events to a <*script*> (see Section 2.9). The different cases of using the <*event-binding*> element are listed below.

Case 1: Binding an event from a source to a recipient bean which implements the appropriate listener interface:
Syntax:

```
  <event-binding [target="alternate-bean"] name="event-set-name" [filter="filter"]>
    <bean ../>
  </event-binding>
```

Semantics: Makes the child bean a listener of "event-set-name" events from the target bean. If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then using that. The filter attribute is unsupported in the current version of BML (it is a place-holder for JDK 1.2 property change and vetoable property change events support in BML). The child bean must implement the necessary listener interface or this operation will fail. Note: the use of

“event-set-name” above instead of “event-name” is intentional - the distinction between event set name and event name is subtle in JavaBeans and is best explained by the beans spec or various books.

Examples:

```
<bean class=”java.awt.Button”>
  <event-binding name=”action”>
    <bean source=”myActionHandler”/>
  </event-binding>
</bean>

<bean class=”java.awt.TextField”>
  <event-binding name=”text”>
    <bean source=”myTextHandler”/>
  </event-binding>
</bean>
```

Case 2: Binding an event from a source to some script:
Syntax:

```
<event-binding [target=”alternate-bean”] name=”event-set-name” [filter=”filter”]>
  <script ../>
</event-binding>
```

Semantics: Causes the child script to be invoked when event “event-set-name” fires in the target bean. If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then using that. The filter attribute is used to indicate a specific method in the listener interface (except for property change and vetoable change events where the filter identifies a specific property) via which the event must be received for the script to be invoked. The value of the filter attribute (if any - or null) and the arguments of the method via which the event was delivered are made available to the script using the special names “event:arg0” (for the filter), “event:arg1”, “event:arg2”, etc.. How one accesses these names from within a script is described later in Section 2.9.

In order for this binding to succeed, an appropriate event adapter for the event type must be available via the BML event adapter registry. See Section 4.4 for discussion about event adapters.

Examples:

```
<bean class="java.awt.TextField">
  <event-binding name="action">
    <script>
      <property target=”bml:arg1” name=”source” id=”the-text-field”/>
      <property target="calculator" name="a">
        <cast class=”int”>
          <property target=”the-text-field” name=”text”/>
        </cast>
      </call-method>
    </script>
  </event-binding>
</bean>

<bean class=”java.awt.Button”>
  <event-binding name=”action”>
    <script>
      <call-method target=”juggler” name=”start”/>
    </script>
  </event-binding>
</bean>
```

```
<bean class="java.awt.Frame">
  <event-binding name="window" filter="windowClosing">
    <script>
      <call-method target="class:java.lang.System" name="exit">
        <cast class="int"><string>0</string></cast>
      </call-method>
    </script>
  </event-binding>
</bean>
```

2.6 Calling Bean Methods

The <*call-method*> element can be used to call methods on beans. Calling a method involves identifying the name of the method to call, the bean on which to call that method and the arguments to the method. The actual method to call is determined using the types of the arguments and applying the Java language method resolution algorithm to find the precise method. (The return type is not necessary to identify the method as Java does not allow overloading of return types.)

Syntax:

```
<call-method [target="alternate-bean"] name="method-name" [id="name"]>
  ... zero or more bean, string, field, property, call-method, cast or script
      elements ...
</call-method>
```

Semantics: Calls the named method on the target bean. If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then using that. The sequence of arguments to the call is created by evaluating all the child elements in order. The signature of the method to search for is defined by the types of the arguments. The method itself is determined using the Java language method resolution logic to find the method with the best matching signature in the target bean's class. The static methods can be invoked via <*call-method*> by setting the target object to the class object of the desired class (see Section 2.1.2). Return bean of the method call (if any) is the value of this element. If the id attribute is present, then the return value is registered in the object registry with that name.

Examples:

```
<bean class="java.awt.Frame">
  <call-method name="pack"/>
  <call-method name="show"/>
</bean>

<bean class="java.awt.Panel">
  <call-method name="setBounds">
    <cast class="int"><string>100</string></cast>
    <cast class="int"><string>100</string></cast>
    <cast class="int"><string>20</string></cast>
    <cast class="int"><string>50</string></cast>
  </call-method>
</bean>
```

2.7 Explicit Type Conversions

As has been mentioned above, BML's <*property*> and <*field*> elements may implicitly invoke a type convertor to convert one type to another. The <*cast*> element allows one to explicitly convert one type to another. The <*cast*> may be purely declarative (i.e., simply a relabelling of the bean to be considered to be of one of its supertypes instead of its type) or actual (where a type conversion is performed).

Syntax:

```
<cast class="class-to-convert-to" [value="string-to-convert"]>
  [... a bean, string, field, property, call-method, cast or script element ...]
```

```
</cast>
```

Semantics: If no content is found and if the value attribute is missing, then this is the same as "(class-to-convert-to)null"; i.e., the null value of the appropriate type. If the "value" attribute is present, then a string bean of that value is type converted to the target class (actually or by re-labeling if strings are assignable to the target class). If no value attribute is present and if a child element is found, then the bean resulting from evaluating that element is type converted to the target class (actually or by re-labeling if the value is assignable to the target class). A common use of the <*cast*> element is to affect the method selection process of <*call-method*>. BML uses a registry of available type convertors (*TypeConvertorRegistry*) which provides a reference to a *TypeConvertor* object that can handle the necessary conversion. The convertor is dynamically invoked with the value to be converted and the result used instead. Type convertors and the type convertor registry are discussed in Section 4.3.

Examples:

```
<bean source="class:java.lang.System">
  <call-method name="getProperty">
    <string>user.name</string>
    <cast class="java.lang.String"/>
  </call-method>
</bean>

<bean class="java.awt.Panel">
  <call-method name="setBounds">
    <cast class="int" value="100"/>
    <cast class="int"><string value="100"/></cast>
    <cast class="int"><string>20</string></cast>
    <cast class="int" value="50"/>
  </call-method>
</bean>
```

2.8 Creating Bean Hierarchies

The <*add*> element is used to create a hierarchy or collection of beans. The <*add*> element abstracts the process of creating bean hierarchies by allowing the use of a single element to "add" a contained bean to some container bean. The motivation for the <*add*> element came from the importance of hierarchies in user interface beans. However, the concept is well defined for any bean which may serve as a container of some sort for other beans, including vectors, hashtables, as well as visual containers such as java.awt.Panel.

The generic "add" concept is implemented by a registry of type-specific adders (see Section 4.4).

Syntax:

```
<add [target="alternate bean"]>
  ... one or more bean, string, field, property, call-method, cast or script
      elements ...
</add>
```

Semantics: Adds one item to a container bean. If the target attribute is present, then the operation is performed on the bean with that name instead of the default (i.e., containing) bean by first looking up that name in the object registry and then adding to that bean. An *Adder* is searched for in the *AdderRegistry* to implement the task of "adding" to the container bean. The *Adder* is searched for using the type of the container bean. The semantics of the children of <*add*> are defined by the specific adder. Adders and the adder registry are discussed in Section 4.4.

Examples:

```
<bean class="java.awt.Panel">
  <add>
    <bean class="java.awt.Button"/>
  </add>
```

```
</bean>

<bean class="java.awt.Panel">
  <property name="layout">
    <bean class="java.awt.BorderLayout"/>
  </property>
  <add>
    <bean class="java.awt.Button"/>
    <string>Center</string>
  </add>
</bean>

<bean class="java.util.Hashtable" id="aliasTable">
  <add>
    <string>ls</string>
    <string>ls -Fa</string>
  </add>
  <add>
    <string>j</string>
    <string>jobs -l</string>
  </add>
</bean>
```

2.9 DEFINING SCRIPTS

The <*script*> element can be used to define a sequence of statements in BML or in some supported scripting language. To add support for languages other than BML, visit http://www.alphaWorks.ibm.com/tech/bsf. For each language desired, 2 things are necessary: the driver, and the language jar. For convenience (and for historical reasons), some of the documentation in this guide refers to JavaScript and NetRexx. Support for these languages is not built in; files will still need to be retrieved from the BSF (Bean Scripting Framework) site (see URL above).

Evaluating a <*script*> results in a bean. For BML scripts, the bean is the value of the last element contained in the script. For a script in a non-BML language, the return value is defined by that language. Typically, the return value is the result of evaluating the last statement of the script.

Syntax:

```
<script [language="script-language]>
  [<args>
    script-arg-1
    script-arg-2
    ...
    script-arg-n
  </args>]
  ... content based on value of script-language attribute ...
</script>
```

Semantics: Defines a new script in the indicated language. The valid values for the language attribute are "bml", or the name of another supported scripting language (such as "javascript" or "netrexx"). If the language attribute is missing, the default value of "bml" is used. Scripts can be given arguments, which are the beans produced by evaluating BML elements in the <*args*> element. If an args element is present as the first element child of the script element, then each argument is evaluated using as the default target the bean containing the script element. Then, the values of the arguments are registered using the special names "script:arg0", "script:arg1", etc. in a special registry that is available to the script. How one accesses these named objects within the script is dependent on the scripting language used; see the following table for details.

Language	Method of Access
bml	Use BML's bean lookup mechanism (i.e., <bean source="script:arg0"> etc.).

javascript	Use the special global object “**bsf**” to access them by calling the method “lookupBean” on that object, giving the name of the bean as an argument. That is, the bean named “script:arg0” would be accessed in JavaScript using ‘bsf.lookupBean (“script:arg0”)’ etc..
netrexx	Use the special global object “**bsf**” to access them by calling the method “lookupBean” on that object, giving the name of the bean as an argument. That is, the bean named “script:arg0” would be accessed in NetRexx using ‘bsf.lookupBean (“script:arg0”)’ etc..

Note: All script elements specifying the “javascript” language share a single global context. That is, anything created within a JavaScript script will be available to later JavaScript scripts. This is not the case for BML and NetRexx.

For scripts written in BML (ie., language=”bml”), the script element basically defines a nested, local namespace for the object registry. That is, if an object is looked up from within the script, it is first searched for in the script’s object registry. If not found, then it is searched for in the containing element’s object registry and so on. However, any names registered within a script are purely local - one cannot register a name within a script so that it is visible outside of the context of the script.

For non-BML languages, the script itself is the content of <*script*> beginning immediately following <*args*> (if it exists). The content of <*script*> is mixed XML content; i.e., it consists of any number of inter-mixed CDATA sections and text. If the scripting language is non-XML (e.g., JavaScript) and the script contains any XML-illegal characters (such as “<”), then the script code must be enclosed in an XML CDATA section. For non-XML scripts, the practice of always enclosing the script in a CDATA section is probably a safe practice, as this avoids the useless, cryptic messages generated by XML parsers when XML-sensitive characters are found in non-XML text.

There is an important semantic difference between a <*script*> contained directly within an <*event-binding*> and one contained elsewhere. Within an <*event-binding*>, it is processed at the time of the event-firing (i.e. its execution is delayed). In other cases, it is processed immediately (i.e. its execution is immediate).

Examples:

```
<bean class="java.awt.TextField" id="TF-prod">
  <event-binding name="propertyChange" filter="prod" target="calculator">
    <script>
      <property target="TF-prod" name="text">
        <property target="bml:arg1" name="newValue"/>
      </property>
    </script>
  </event-binding>
</bean>

<bean class=”java.awt.TextField”>
  <property name=”text”>
    <script language=”javascript”>5*10</script>
  </property>
</bean>

<bean class=”java.awt.TextField” id=”tf”>
  <event-binding name=”action”>
    <script language=”javascript”>
      <args><property target=”tf” name=”text”/></args>
      <![CDATA[
      currentText = bsf.lookupBean (“script:arg0”);
      eventObj = bsf.lookupBean (“event:arg1”);
      eventObj.getSource().setText (currentText + “ “ + currentText);
      ]]>
    </script>
```

```
  </event-binding>
</bean>

<bean class="java.awt.TextField" id="tf">
  <event-binding name="action">
    <script language="netrexx">
      <args><property target="tf" name="text"/></args>
      <![CDATA[
      currentText = java.lang.String bsf.lookupBean("script:arg0");
      eventObj = java.awt.event.ActionEvent bsf.lookupBean("event:arg1");
      sourceField = java.awt.TextField eventObj.getSource();
      sourceField.setText(currentText " " currentText);
      ]]>
    </script>
  </event-binding>
</bean>
```

2.10 BML Processing Instruction

BML processors support a processing instruction that may be used to instruct the processor to load components such as type convertors, adders and event adapters (see Section 4.4 for details of these).

Syntax:

```
<?bml register name-of-registerable-class?>
```

Semantics: The class of name "name-of-registerable-class" must implement the interface `com.ibm.bml.BMLSelfRegisterable`, which is defined as follows:

```
public interface BMLSelfRegisterable {
  public void register (BMLEnvironment env);
}
```

The BML processor instantiates this class and then calls its register method with the processor's environment (see Section 4.1) as an argument. The body of the register method can register anything into that environment. This processing instruction may appear either within the document element or outside of it.

Examples:

```
<?bml register RegisterMyStuff?>

<bean class="com.sun.java.swing.JFrame">
  ... bml code that assumes the things registered by the above ...
</bean>
```

where the RegisterMyStuff class is as follows:

```
public class RegisterMyStuff implements BMLSelfRegisterable {
  public void register (BMLEnvironment env) {
    Adder ad = new Adder () {
      .. a new adder ..
    };
    env.adderRegistry.register (..., ad);
  }
}
```

3. BML Processing Model

A BML script is an XML document whose document element is a <*bean*> element. Arbitrary configurations of this and other beans can be expressed within this element using combinations of the above elements. When a BML script is evaluated in textual order, the bean defined by the document element is produced and configured according to the specifications inside that element. The specifications can of course include creating other beans and adding to container beans.

We have two implementations of BML: the *player* and the *compiler*. The player uses Java reflection to evaluate a BML script at startup-time of an application. The compiler is a static tool that generates Java code that, at startup-time, will produce a bean configuration equivalent to that described in the script. During compilation, the compiler may temporarily instantiate some beans to learn their properties, events and methods, but these beans are discarded afterwards.

In JavaBeans, configuration-time is when a bean can put up its configuration user interfaces etc.. BML views this configuration time as a process that occurs in some environment which produces BML to express a desired configuration. Per-bean configuration can also be saved into serialization (.ser) files. If BML is generated by some batch process, then the configuration-time of beans may never occur.

BML views the process of executing the bean configuration specification as processing-time. This always occurs at startup in BML-based applications and once configuration is complete, run-time proper starts. How the configuration is actually implemented is different depending on whether the player or the compiler does the work: If it is the player, then it is done using reflection. If it is the compiler, then the generated Java code does it.

3.1 BML Processor

Both the player and the compiler inherit from a common base class called `com.ibm.bml.BMLProcessor`. This allows one to treat both the processors in a similar manner, except of course each provides different customizable properties which must be dealt with as the specific type of processor (`com.ibm.bml.player.BMLPlayer` or `com.ibm.bml.compiler.BMLCompiler`). The DemoMgr class in the demos/applettool directory provides an example of using the two in a unified manner. The BMLProcessor class is defined as follows:

```
public abstract class BMLProcessor {
  protected BMLEnvironment env;

  public BMLProcessor () {
    env = new BMLEnvironment ();
  }

  public BMLProcessor (BMLEnvironment env) {
    this.env = env;
  }

  public void setBMLEnvironment (BMLEnvironment env) {
    this.env = env;
  }

  public BMLEnvironment getBMLEnvironment () {
    return env;
  }

  public abstract Object processDocument (Document doc) throws BMLException;

  public abstract Object processDocument (Document doc, URL contextURL)
                                                          throws BMLException;
}
```

A `com.ibm.bml.BMLEnvironment` (see below) can be passed as a constructor argument to a `BMLProcessor`; if the no-arg constructor is used, a default `BMLEnvironment` will be created.

3.1.1 The contextURL

In addition to being given a `org.w3c.dom.Document` to process, a `BMLProcessor` can optionally be given a contextURL from which to resolve relative URLs (effectively a document base). The contextURL is updated to reflect the file currently being processed.

Example:
If file `http://bml.watson.ibm.com/app.bml` is being processed, the contextURL is: `http://bml.watson.ibm.com`. If this file refers to `http://w3.ibm.com/myMenuBar.bml`, the contextURL (while processing `myMenuBar.bml`) will be `http://w3.ibm.com`. This means that any references within `myMenuBar.bml` will be resolved relative to `http://w3.ibm.com`, not `http://bml.watson.ibm.com`.

For more information on the way a `BMLProcessor` will resolve URL and file locations, see the section on creating <*bean*>s.

Note: Calling `processDocument(doc)` is equivalent to calling `processDocument(doc, null)`.

In the next section we discuss the processing-time and run-time environments of BML.

4. BML Processing- and Run-time Environments

The processing-time and run-time components of BML consist of a set of registries that provide references to various objects and services. In this section we describe each component and its role.

4.1 BML Environment

The entire environment of a BML processor (the player or the compiler) is represented by the class `com.ibm.bml.BMLEnvironment`. This class contains references to each of the registries (object registry, type convertor registry, adder registry and event adapter registry), the BSFManager (used for scripting language support), the class loader and the XMLParserLiaison used throughout BML. The `BMLEnvironment` is a property of a BML processor and can be configured fully within an instance of a processor.

```
public class BMLEnvironment implements Cloneable {
  public ObjectRegistry objectRegistry =
    BMLGlobalEnvironment.objectRegistry;
  public TypeConvertorRegistry typeConvertorRegistry =
    BMLGlobalEnvironment.typeConvertorRegistry;
  public AdderRegistry adderRegistry =
    BMLGlobalEnvironment.adderRegistry;
  public EventAdapterRegistry eventAdapterRegistry =
    BMLGlobalEnvironment.eventAdapterRegistry;
  public BSFManager BSFmgr = (BMLGlobalEnvironment.BSFmgr == null)
                             ? new BMLBSFManager (this)
                             : BMLGlobalEnvironment.BSFmgr;
  public ClassLoader classLoader = (BMLGlobalEnvironment.classLoader == null)
                                   ? getClass ().getClassLoader ()
                                   : BMLGlobalEnvironment.classLoader;
  public XMLParserLiaison parserLiaison = BMLGlobalEnvironment.parserLiaison;
}
```

The default environment is represented by the class `com.ibm.bml.BMLGlobalEnvironment`, which has (public) static fields containing the initial values for the items in the `BMLEnvironment` class. One may change the

default global environment which every new instance of `BMLEnvironment` (and hence every new player / compiler instance) inherits by assigning new values to the public fields of the `BMLGlobalEnvironment` class:

```
public class BMLGlobalEnvironment {
  public static ObjectRegistry objectRegistry = new ObjectRegistry ();
  public static TypeConvertorRegistry typeConvertorRegistry =
    new TypeConvertorRegistryImpl ();
  public static AdderRegistry adderRegistry =
    new AdderRegistryImpl ();
  public static EventAdapterRegistry eventAdapterRegistry =
    new EventAdapterRegistryImpl ();
  public static BSFManager BSFmgr = null;
  public static ClassLoader classLoader = null;
  public static XMLParserLiaison parserLiaison = new XMLParserLiaisonImpl ();
}
```

Each registry is defined by an interface (except the ObjectRegistry, which is a class) which is used by all of BML to interact with the registry. The BML distribution comes with default implementations of these registries, but the defaults can be replaced easily to provide customized functionality. The default value of null for the class loader property effectively makes BML use the system class loader for loading classes. In the rest of this section, we describe each of the registries used in BML, and the XMLParserLiaison interface.

4.2 Object Registry

The `objectRegistry` (`com.ibm.bml.ObjectRegistry`) provides a string name to object reference mapping. If a parent `ObjectRegistry` is provided at construction time, lookups will cascade upwards. Registrations, on the other hand, are always done locally. This registry serves three primary purposes: communication of object handles between different parts of BML scripts, communication of object handles between Java code and BML scripts, and communication of object handles between BML and scripts defined in scripting languages such as JavaScript or NetRexx. The `ObjectRegistry` is defined as follows:

```
public class ObjectRegistry {
  Hashtable reg = new Hashtable ();
  ObjectRegistry parent = null;
  public ObjectRegistry () {
  }
  public ObjectRegistry (ObjectRegistry parent) {
    this.parent = parent;
  }
  public void register (String name, Object obj) {
    reg.put (name, obj);
  }
  public void unregister (String name) {
    reg.remove (name);
  }
  public Object lookup (String name) throws IllegalArgumentException {
    Object obj = reg.get (name);
    if (obj == null && parent != null) {
      obj = parent.lookup (name);
    }
    if (obj == null) {
      throw new IllegalArgumentException ("object '" + name + "' not in registry");
    }
    return obj;
  }
}
```

One can replace the default object registry simply by extending this class and updating the `BMLEnvironment` property of the BML processor as follows:

```
BMLProcessor p = new BMLPlayer (); // or new BMLCompiler ();
BMLEnvironment env = p.getBMLEnvironment ();
env.objectRegistry = new ObjectRegistry () {
  public void register (String name, Object obj) {
    ... do whatever you want to do to register the obj with the given name ...
  }
  public void lookup (String name) {
    ... do whatever you want to do to lookup the name in the registry ...
  }
};
// now use this processor
p.processDocument (doc);
```

4.3 Type Convertors and Type Convertor Registry

Type convertors are objects that know how to convert an object of one type to an object of another type. Type convertors implement the `com.ibm.bml.TypeConvertor` interface which is defined as follows:

```
public interface TypeConvertor {
  public Object convert (Class from, Class to, Object obj);
  public String getCodeGenString ();
}
```

The convert method is given the object to convert and the source and target types of the conversion and must return a new object of the target type. The getCodeGenString method must return a piece of Java code (a full method except without the name of the method) that can be placed into the generated code by the compiler when a BML document is compiled to not use the BML registries at runtime. An example convertor to convert from any object type to String is shown below:

```
TypeConvertor tc = new TypeConvertor () {
  public Object convert (Class from, Class to, Object obj) {
   return (obj == null) ? "(null)" : obj.toString ();
  }

  public String getCodeGenString () {
    return "(Class from, Class to, Object obj) {\n" +
           "return (obj == null) ? \"(null)\" : obj.toString ();\n" +
           "}";
  }
};
// register the new type convertor to my player/compiler
BMLProcessor p = new BMLCompiler ();
BMLEnvironment env = p.getBMLEnvironment ();
env.typeConvertorRegistry.register (Object.class, String.class, tc);
```

The *type convertor registry* (`com.ibm.bml.TypeConvertorRegistry`) provides a registration and look up service for type convertors. The TypeConvertorRegistry interface is defined as follows:

```
public interface TypeConvertorRegistry {
  public void register (Class from, Class to, TypeConvertor convertor);
  public TypeConvertor lookup (Class from, Class to) throws BMLException;
}
```

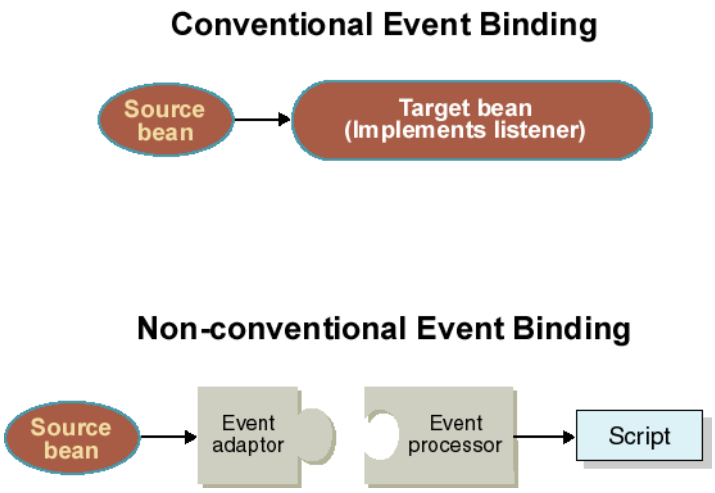
The built-in implementation of this registry is used by default by BML processors. This implementation uses a simple hashtable as its registry. The built-in registry comes with convertors for the following types:

- all primitives types to primitive object wrapper types (e.g., int to/from java.lang.Integer)
- strings to all primitive types (e.g., java.lang.String to float)
- any type to java.lang.String (which is used as a backup convertor if the target type is string and if no other convertor is available)
- java.lang.String to java.awt.Font
- java.lang.String to java.awt.Color

One can replace the default type convertor registry simply by providing another implementation of this interface and updating the `BMLEnvironment` property of the BML processor as done in Section 4.2 for the object registry.

4.4 Event Adapters, Event Processors and Event Adapter Registry

BML supports "extended" event bindings to an arbitrary script *without* making any assumptions about the types of events that may be thrown by beans. This capability is supported by event adapters, event processors and an event adapter registry. Basically, the model consists of an event-type specific adapter that receives the event from the source, forwards it to a generic event processor which then runs the script. Event-type specific event adapters are located from the event adapter registry. The figure below illustrates the standard and extended binding architecture.



Event adapters must implement the `com.ibm.cs.event.EventAdapter` interface to be part of the BML event architecture. This interface is defined as follows:

```
public interface EventAdapter {
  public void setEventProcessor (EventProcessor eventProcessor);
}
```

A base implementation of this interface is available in `com.ibm.cs.event.EventAdapterImpl` that event adapters may choose to extend. The code for EventAdapterImpl is as follows:

```
public class EventAdapterImpl implements EventAdapter {
  protected EventProcessor eventProcessor;

  public void setEventProcessor (EventProcessor eventProcessor) {
    this.eventProcessor = eventProcessor;
  }
}
```

For each type of event (actually, for each listener type), an event adapter must be implemented and available from the event adapter registry. For example, the following event adapter is included in the default event adapter registry for adapting java.awt.event.KeyEvent events to the BML event architecture:

```
public class java_awt_event_KeyAdapter extends EventAdapterImpl
                                        implements KeyListener {
  public void keyTyped (KeyEvent e) {
    eventProcessor.processEvent ("keyTyped", new Object[]{e});
  }
  public void keyPressed (KeyEvent e) {
    eventProcessor.processEvent ("keyPressed", new Object[]{e});
  }
  public void keyReleased (KeyEvent e) {
    eventProcessor.processEvent ("keyReleased", new Object[]{e});
  }
}
```

When the BML runtime creates event adapters and adds them as listeners to the event sources, it tells the adapter what event processor to forward (delegate) events to. Event processors are the entry point to the BML runtime and are responsible for delivering the event to the intended recepient script. The `com.ibm.cs.event.EventProcessor` interface is defined as follows:

```
public interface EventProcessor {
  public void processEvent (String filter, Object[] eventInfo);
  public void processExceptionableEvent (String filter, Object[] eventInfo)
                                         throws Exception;
}
```

When an event adapter receives an event from an event source, it delegates the event to its event processor using one of the above two methods. If the event method may throw an exception, then it should delegate it via the `processExceptionableEvent` method, otherwise via the `processEvent` method. The filter argument is in general the name of the method via which the event was received. The exception to this is for `java.beans.PropertyChangeListener` and `java.beans.VetoableChangeListener` event listener types, where the filter is the name of the property. The BML player uses a single event processor that actually delivers the event to a script and runs the script. The compiler can generate customized event processors that perform that task, or it may in fact completely bypass this entire mechanism and simply generate customized event adapters that directly deliver the event to the user's script. If the compiler is run in the "generate BML-independent code" mode, then it in fact does this. However, when it generates code that uses the BML runtime, it does use this architecture. The latter is necessary when the generated code uses the BML runtime for its operation (for example, the applettool demo in the demos directory).

The *event adapter registry* (`com.ibm.bml.EventAdapterRegistry`) provides a registration and look up service for event adapters. The EventAdapterRegistry interface is defined as follows:

```
public interface EventAdapterRegistry {
  public void register (Class listenerType, Class eventAdapterClass);
  public Class lookup (Class listenerType) throws BMLException;
}
```

The built-in implementation of this registry is used by default by BML processors. This implementation uses a simple hashtable as its registry. The lookup algorithm of this implementation first looks for the adapter in its own hashtable. If not found, then it tries to load an event adapter from the `com.ibm.cs.event.adapters` package by converting the event listener type's package qualified name using the following rules:

- the word "Listener" is dropped from the package qualified name
- each "." character (package level separator) is replaced with the "_" character

- the string "`com.ibm.cs.event.adapters`" is prepended to the result
- the string "Adapter" is appended to it

For example, for the `java.awt.event.KeyListener` event listener type, the name `com.ibm.cs.event.adapters.java_awt_event_KeyAdapter` is generated. The resulting name is used as the name of the event adapter class and the class loader is used to load that class. If the loading fails, then the event adapter is deemed unavailable and an exception is thrown. The bmlextensions.jar file contains adapters that conform to these rules for the following event types:

- all `java.awt.event.*` event listener types
- `java.beans.PropertyChangeListener` and `java.beans.VetoableChangeListener` event listener types

One can replace the default event adapter registry simply by providing another implementation of this interface and updating the `BMLEnvironment` property of the BML processor as done in Section 4.2 for the object registry.

4.4.1 Dynamic Event Adapter Generation

This extension eliminates the need to write event adapters by automatically generating the class files (not via compilation from source, but directly) for event adapters on demand. Since the BML event adapaters are relatively straight forward and highly stylized (see the architecture discussion above), it is possible to easily generate the bytecodes for event adapters directly without going through a compilation stage. The dynamic event adapter generator performs this function.

The dynamic event adapter genrerator is made available to BML users as an alternate event adapter registry. The demo driver includes this functionality if you use the "-dynamic" switch. In that case, the BML processor's environment's event adapter generator is changed to a new instance of `com.ibm.bml.extensions.generator.DynamicEventAdapterGenerator`. This registry automatically invokes the event adapter generator to produce a new event adapter class if the event listener type has not been seen previously.

4.5 Adders and Adder Registry

Adders and the adder registry are the components that implement the containment model enabled by the <*add*> element in BML. An adder is an object that implements the `com.ibm.bml.Adder` interface which is defined as follows:

```
public interface Adder {
  public void add (Class parentClass, Object parent, Object[] args);
  public String getCodeGenString ();
}
```

An adder is an object that knows how to add to some type(s) of container bean(s). The add method is invoked with the type of container, the container bean as well as the beans resulting from evaluating the children elements of the <*add*> element. The adder defines the semantics of the argument beans (i.e., those contained within the <*add*> element) and implements actual task of "adding" to the container. For example, for a Vector container, the adder would require exactly one bean to be present inside the <*add*> element and would add that bean to the vector using the "addElement" method. For a Swing JFrame container, the adder would call the "getContentPane" method on the JFrame and then add to the content pane by calling the "add" method on that with one or two arguments depending on the contents of the <*add*> element. Such a Swing adder is shown below:

```
Adder ad = new Adder () {
  public void add (Class parentClass, Object parent,Object[] args) {
    com.sun.java.swing.RootPaneContainer rpc =
      (com.sun.java.swing.RootPaneContainer) parent;
    Container c = rpc.getContentPane();

    if (args.length == 2) {
```

```
      c.add ((Component)args[0], args[1]);
    } else if (args.length == 1) {
      c.add ((Component)args[0]);
    } else {
      System.err.println ("ERROR:" + " RootPaneAdder takes 1" + " or 2 args");
    }
  }

  public String getCodeGenString () {
    return "(Class parentClass, Object parent,Object[] args)\n" +
           "{\n" +
           "com.sun.java.swing.RootPaneContainer rpc =\n" +
           "(com.sun.java.swing.RootPaneContainer)parent;\n" +
           "Container c = rpc.getContentPane();\n" +
           "\n" +
           "if (args.length == 2)\n" +
           "{\n" +
           "c.add((Component)args[0], \n" +
           "args[1]);\n" +
           "}\n" +
           "else if (args.length == 1)\n" +
           "{\n" +
           "c.add((Component)args[0]);\n" +
           "}\n" +
           "else\n" +
           "{\n" +
           "System.err.println (\"ERROR:\" + \" RootPaneAdder takes 1\"" +
           "+ \" or 2 args\");\n" +
           "}\n" +
           "}";
  }
});
// register the new adder to my player/compiler
BMLProcessor p = new BMLCompiler ();
BMLEnvironment env = p.getBMLEnvironment ();
env.adderRegistry.register (com.sun.java.swing.RootPaneContainer.class, ad);
```

This adder can now be used with any Swing container type that implements the RootPaneContainer interface. These include JFrame, JTabbedPane, and JPanel.

The *adder registry* (`com.ibm.bml.AdderRegistry`) provides registration and look up service for adders. The AdderRegistry interface is defined as follows:

```
public interface AdderRegistry {
  public void register (Class parentClass, Adder adder);
  public Adder lookup (Class parentClass) throws BMLException;
}
```

When processing an <*add*> element, the target bean's type is used as the key to find the adder. If an adder is not found for that specific type, the search is continued up the type hierarchy of the container. For example, if the container's type is java.awt.Frame, then if an adder is not found for java.awt.Frame, then the search is continued for java.awt.Window and then java.awt.Container, the super classes of java.awt.Frame. If the root class (java.lang.Object) is reached before an adder is located, an exception is thrown to indicate the absence of an adder.

The Java type system is more complicated however - while a class may inherit code from exactly one other class, it may inherit behavior from any number of other types (by implementing multiple interfaces). The adder search

algorithm in the built-in adder registry implementation first searches for adders in interfaces implemented by the container type and then in the superclass. The algorithm is as follows:

```
- is there an adder for container type? If yes, return that
- for each interface the container implements,
  - recursively look for an adder for the interface type
  - if found done, else look at next interface type
- end for
- recursively look for an adder for container's superclass type
- if after all recursive processing an adder is not found, throw exception
```

The built-in implementation of the adder registry is used by default by BML processors. This implementation uses a simple hashtable as its registry. The built-in registry comes with adders for the following container types:

- java.awt.Container
- java.util.Dictionary (and hence for types such as java.util.Hashtable)
- java.util.Vector

One can replace the default adder registry simply by providing another implementation of this interface and updating the BMLEnvironment property of the BML processor as done in Section 4.2 for the object registry.

4.6 XMLParserLiaison

XML parser liaisons are used throughout BML to retrieve a `org.w3c.dom.Document` from a `Reader`. XML parser liaisons implement the `com.ibm.cs.util.XMLParserLiaison` interface which is defined as follows:

```
public interface XMLParserLiaison {
  public Document readStream (String sourceDesc, Reader reader);
}
```

The default implementation, which uses IBM's XML4J parser, is defined as follows:

```
public class XMLParserLiaisonImpl implements XMLParserLiaison {
  public Document readStream (String sourceDesc, Reader reader) {
    return new com.ibm.xml.parser.Parser (sourceDesc).readStream (reader);
  }
}
```

This implementation is used by default by BML processors. One can replace the default XML parser liaison (so as to use a different XML parser) simply by providing another implementation of this interface and updating the `BMLEnvironment` property of the BML processor as done in Section 4.2 for the object registry.

5. Embedding a BML Processor in an Application

BML processors have been carefully designed to support embedding into other applications. Both the player and the compiler are well-behaved beans and their run/processing-time contexts are exposed via their `BMLEnvironment` property. The following illustrates how to embed the player (for example) into a Java application:

```
BMLPlayer player = new BMLPlayer ();
// every new instance of a player/compiler gets a new BMLEnvironment which
// gets defaults from the BMLGlobalEnvironment class (see Section 4.1)

// modify the environment if necessary
BMLEnvironment env = player.getBMLEnvironment ();
env.eventAdapterRegistry = new DynamicEventAdapterRegistry ();
env.objectRegistry = new ObjectRegistry () {
  // implement register() and lookup()
}
```

```
// pre-register any special things into the various registries
env.objectRegistry.register (“foo”, ...);

// process BML documents:
Document d = call-xml-parser (XML URL);
Object o = player.processDocument (d [, contextURL]);

// process object o
do-something-with-result (o);
```

6. BML Player and Compiler Beans

The player and compiler are both standard beans which inherit from the `com.ibm.bml.BMLProcessor` base class (see Section 3). In this section we list the properties of these beans.

6.1 BML Player

Property Name	Type	Accessibility	Description
BMLEnvironment	BMLEnvironment	Read/Write	The runtime environment of the player. Default is a new instance of BMLEnvironment which gets default registry etc. values from the BMLGlobalEnvironment class.

6.2 BML Compiler

Property Name	Type	Accessibility	Description
BMLDep	boolean	Read/Write	Turns on/off dependency of generated code on BML runtime. Default is on.
BMLEnvironment	BMLEnvironment	Read/Write	The runtime environment of the compiler. Default is a new instance of BMLEnvironment which gets default registry etc. values from the BMLGlobalEnvironment class.
className	String	Read/Write	Fully-qualified name of Java class to generate. Default is “Test”.
codeDesc	String	Read/Write	Name to use in status / error messages. Default is <STDIN>.
formatOutput	boolean	Read/Write	Turns on/off formatting of generated Java code (e.g. word-wrapping, indentation). Default is on.
methodName	String	Read/Write	Name of the method to execute in the generated code to get the bean described by the BML script (the “service method”). Default is “exec”.
outputWriter	Writer	Write	Sets the target for the generated code stream. Default is java.lang.System.out.
showStatus	boolean	Read/Write	Turns on/off status reporting during compilation. Default is on.
useDefaultEnvironment	boolean	Read/Write	Turns on/off the assignment of the global environment to the BMLEnvironment in the generated code (when BML dependent code is generated). Default is on.
errorCount	int	Read	Number of errors during last compilation.